Karlsruher Institut für Technologie

Software Design and Quality

Coupling Architectural Analyses and Static Source Code Analyses for Detecting Security Vulnerabilities

Zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften

von der KIT-Fakultät für Informatik
des Karlsruher Instituts für Technologie (KIT)

genehmigte

Dissertation

von

Frederik Reiche

Tag der mündlichen Prüfung: 11. Mai 2026
Erster Gutachter: Prof. Dr. Robert Heinrich, Universität Ulm
Zweiter Gutachter: Prof. Dr. Hans Vangheluwe, University of Antwerp

Abstract

In this dissertation, we investigate the coupling of architectural security analyses with static source code security analyses to detect architectural vulnerabilities that arise from an implementation that is non-compliant with the architectural specification. We present properties of coupling approaches suitable for achieving this goal (called coupling design), two coupling approaches that instantiate this design, and an approach to relate static source code analyses to design-level security domain-specific languages.

Most modern software systems process or store sensitive information, and are interconnected with other systems. Consequently, they must be designed and implemented securely to prevent attackers from exploiting vulnerabilities in the system and to compromise its security. Compromising the security can have severe consequences for individuals and companies, such as financial loss, impersonation, or loss of trust. Vulnerabilities in safety-critical systems, such as mobility systems, energy supply or healthcare systems, present particular alarming cases for misuse.

These insights motivated source code security analyses to detect vulnerabilities in the implementation. Furthermore, architectural security analyses have been developed that analyze the architectural design to detect vulnerabilities before they manifest in the implementation. However, research has shown that evolutionary changes can result in non-compliance between the architectural design and the implementation. Non-compliance poses a significant challenge for architectural analyses, because they may miss vulnerabilities that actually exist in the final system. This leads to the challenge that software engineers may wrongly assume the system to be secure based on the architectural analysis results, while an attacker can exploit vulnerabilities in the deployed system.

In this dissertation, we address the challenge of missed architectural vulnerabilities due to non-compliant implementations by coupling architectural analyses with static source code analyses. We investigate suitable properties of coupling approaches, relations to bridge the gap between architectural security and implementation-level security, and mechanisms to enable the information exchange between architectural analyses and source code analyses. Our first contribution, the coupling design, provides reusable knowledge on properties suitable to couple architectural analyses with source code analyses. The second contribution comprises two coupling approaches addressing two types of static source code analyses and different types of security-related properties. Both coupling approaches instantiate our coupling design, define information necessary for the coupling, define how to relate this information in the analyses, and define how to enable the information exchange between the analyses. Our third contribution is SecLan, an approach to determine relations between the vulnerabilities detectable by source code analyses and security

properties specified in design-level domain-specific languages based on a conceptual model.

Our contributions are empirically evaluated using evaluation scenarios for the coupling design and coupling approaches, and expert surveys and expert interviews for SecLan. The results of our evaluation show that the information defined by our coupling approaches are covered by the analyses, and that the relations defined exist in successful couplings. The evaluation of accuracy shows that the coupling approaches can detect architectural vulnerabilities arising from non-compliant implementations that would have been missed without the coupling. However, our results also show that one coupling approach has further potential for improvement due to limited accuracy, while the other coupling approach exposes high accuracy. The results of our evaluation of SecLan show that the conceptual model is correct and complete, and that the relations calculated by SecLan are valid. Most notably, our results show that also security experts are not always aware of the relations between source code analysis results and security properties specified in security-related system designs, highlighting the necessity for SecLan and our coupling approaches.

Overall, our contributions allow software engineers to detect architectural vulnerabilities arising from non-compliant implementations, and to systematically relate source code analyses to architectural analyses.

Zusammenfassung

In dieser Dissertation untersuchen wir die Kopplung von architekturbasierten Sicherheitsanalysen mit statischen Quelltextsicherheitsanalysen, um Sicherheitsschwachstellen in der Softwarearchitektur zu erkennen, die aus einer Implementierung entstehen, die nicht mit der Architekturspezifikation übereinstimmt. Wir stellen Eigenschaften von Kopplungsansätzen vor, die geeignet sind, dieses Ziel zu erreichen (als Kopplungsentwurf bezeichnet), zwei Kopplungsansätze, die diesen Entwurf instanziieren, sowie einen Ansatz, um statische Quelltextsicherheitsanalysen mit domänenspezifischen Sicherheitssprachen auf Entwurfsebene in Beziehung zu setzen.

Die meisten modernen Softwaresysteme verarbeiten oder speichern sensible Informationen und sind mit anderen Systemen vernetzt. Aus diesem Grund müssen sie sicher entworfen und implementiert werden, um zu verhindern, dass Angreifer Schwachstellen im System ausnutzen und die Sicherheit kompromittieren. Die Kompromittierung der Sicherheit kann schwerwiegende Folgen für Einzelpersonen und Unternehmen haben, zum Beispiel finanzielle Verluste, Identitätsmissbrauch oder Vertrauensverlust. Schwachstellen in der kritischen Infrastruktur, etwa im Mobilitätsbereich, in der Energieversorgung oder im Gesundheitswesen, stellen besonders alarmierende Missbrauchsrisiken dar.

Diese Erkenntnisse motivierten Quelltextsicherheitsanalysen, um Schwachstellen in der Implementierung zu erkennen. Darüber hinaus wurden architekturbasierte Sicherheitsanalysen entwickelt, die den Architekturentwurf analysieren, um Schwachstellen zu erkennen, bevor sie sich in der Implementierung manifestieren. Forschung in diesem Bereich hat jedoch gezeigt, dass evolutionäre Änderungen zu Nichtübereinstimmung zwischen Architekturentwurf und Implementierung führen können. Diese Nichtübereinstimmung stellt eine erhebliche Herausforderung für architekturbasierte Analysen dar, da diese Schwachstellen übersehen können, die im finalen System existieren. Dies führt zu der Herausforderung, dass Softwareingenieure auf Basis der Ergebnisse architekturbasierter Analysen fälschlicherweise annehmen können, ein System sei sicher, während ein Angreifer eine Schwachstelle im bereitgestellten System ausnutzen kann.

In dieser Dissertation adressieren wir die Herausforderung übersehener architekturbezogene Sicherheitsschwachstellen infolge nicht übereinstimmender Implementierungen durch die Kopplung architekturbasierter Analysen mit statischen Quelltextsicherheitsanalysen. Hierfür untersuchen wir geeigneten Eigenschaften von Kopplungsansätzen für die Kopplung architekturbasierter Analysen mit statischen Quelltextsicherheitsanalysen, Beziehungen zur Überbrückung der Lücke zwischen architekturbasierter Sicherheit und Sicherheit der Implementierung, sowie Mechanismen zur Ermöglichung des Informationsaustauschs zwischen architekturbasierten Analysen und Quelltextsicherheitsanalysen.

Unser erster Beitrag, der Kopplungsentwurf, stellt wiederverwendbares Wissen über Eigenschaften bereit, die geeignet sind, architekturbasierte Analysen mit Quelltextsicherheitsanalysen zu koppeln. Der zweite Beitrag umfasst zwei Kopplungsansätze, welche zwei wichtige Typen statischer Quelltextsicherheitsanalysen und unterschiedliche Arten sicherheitsrelevanter Eigenschaften adressieren. Beide Kopplungsansätze instanziieren unseren Kopplungsentwurf, definieren für die Kopplung notwendige Informationen, legen fest, wie diese Informationen in den Analysen in Beziehung gesetzt werden, und definieren, wie der Informationsaustausch zwischen den Analysen ermöglicht wird. Unser dritter Beitrag ist SecLan, ein Ansatz zur Bestimmung von Beziehungen zwischen Sicherheitsschwachstellen die von statischen Quelltextsicherheitsanalysen gefunden werden und Sicherheitseigenschaften welche in domänenspezifischen Sprachen auf Entwurfsebene spezifiziert sind.

Unsere Beiträge werden empirisch evaluiert: mit Evaluierungsszenarien für den Kopplungsentwurf und die Kopplungsansätze sowie mit Expertenumfragen und Experteninterviews für SecLan. Die Ergebnisse unserer Evaluation zeigen, dass die durch unsere Kopplungsansätze definierten Informationen durch die Analysen abgedeckt werden und dass die definierten Beziehungen in erfolgreichen Kopplungen existieren. Die Evaluation der Genauigkeit zeigt, dass die Kopplungsansätze architekturbezogene Sicherheitsschwachstellen erkennen können, die aus nicht übereinstimmenden Implementierungen entstehen und ohne die Kopplung übersehen worden wären. Unsere Ergebnisse zeigen jedoch auch, dass ein Kopplungsansatz aufgrund eingeschränkter Genauigkeit Raum für Verbesserungen bietet, während der andere Kopplungsansatz eine hohe Genauigkeit aufweist. Die Ergebnisse unserer Evaluation von SecLan zeigen, dass das konzeptionelle Modell korrekt und vollständig ist und dass die von SecLan berechneten Beziehungen valide sind. Hervorzuheben ist, dass unsere Ergebnisse darauf hindeuten, dass selbst Sicherheitsexperten sich nicht immer über alle Beziehungen zwischen Quelltextanalyse-Ergebnissen und Sicherheitseigenschaften, die in sicherheitsbezogenen Systementwürfen spezifiziert sind, bewusst sind. Dies unterstreicht die Notwendigkeit von SecLan und unseren Kopplungsansätzen.

Insgesamt ermöglichen unsere Beiträge Softwareingenieuren, architekturbezogene Sicherheitsschwachstellen zu erkennen, die aus nicht übereinstimmenden Implementierungen entstehen, und Quelltextsicherheitsanalysen systematisch mit architekturbasierten Analysen in Beziehung zu setzen.

Danksagung

Diese Arbeit ist das Ergebnis vieler Begegnungen, Diskussionen und Unterstützungen, für die ich von Herzen dankbar bin.

Zunächst möchte ich Robert und Ralf danken, die meine wissenschaftliche Arbeit maßgeblich geprägt und mich die gesamte Zeit hinweg unterstützt haben. Robert danke ich besonders für die unermüdliche Unterstützung bei Paperprojekten, der Dissertation, für die wertvollen Diskussionen, die meine Forschung definiert und bereichert haben, und für das kontinuierliche Feedback. Weiterhin danke ich Ralf für sein außergewöhnlich wertvolles und präzises Feedback, das meine Arbeit in entscheidenden Momenten deutlich verbessert hat, für die Projekte, in denen ich meine Forschung durchführen konnte, und für den großen Einsatz, der es mir ermöglicht hat, diese Arbeit erfolgreich abzuschließen.

Weiterhin danke ich meiner Lebensgefährtin Inga und meinem Sohn Rafael. Danke für eure Unterstützung in meinen Höhen und Tiefen. Ich danke euch dafür, dass ihr es ausgehalten habt, wenn ich euch zeitweise weniger Aufmerksamkeit schenken konnte, als ihr verdient habt. Inga, ich danke dir besonders dafür, dass du mir selbst in sehr anstrengenden Zeiten immer Rückhalt gegeben und meine Macken ausgehalten hast. Rafael, danke für die Ablenkung und das Lachen, das du mir genau dann geschenkt hast, wenn ich es gebraucht habe.

Auch meinen Eltern, Christa und Gerhard, danke ich von Herzen. Ihr habt mich auf meinem gesamten Weg unterstützt und stets an mich geglaubt, sodass ich diesen Punkt in meinem Leben erreichen konnte. Ohne euch wäre ich nicht da, wo ich heute bin. Ebenso danke ich meinem Bruder Christopher, der mir mit wertvollen Ratschlägen immer zur Seite stand, wenn ich sie brauchte.

In dieser Zeit hatte ich das Privileg, viele Kolleginnen und Kollegen kennenzulernen, die meine Arbeit erleichtert und geprägt haben. Dafür möchte ich mich herzlich bedanken. Einige möchte ich besonders hervorheben. Danke an Sandro, der mir bereits in der Masterarbeit den Anstoß gegeben hat, eine Promotion zu beginnen. Jonas, Sophie, Sebastian, Thomas und Florian danke ich für die stets produktive Zusammenarbeit und die unterhaltsamen Gespräche im Rahmen gemeinsamer Projekte in KASTEL. Dadurch wurde aus Projektarbeit auch Vergnügen. Ebenso danke ich Nicolas, Tobias, Kevin, Timur, Larissa und Lars für die vielen unterhaltsamen Gespräche bei einem (oder meist mehreren) Kaffee.

Somit ist es geschafft und dieser Abschnitt meines Lebens beendet. Nochmals allen die mich auf diesem Weg begleitet haben (ob zuvor genannt oder ungenannt): Danke!

Contents

List of Figures

List of Tables

Listings

List of Acronyms

ADL Architecture Description Language . . . . . 17 f., 29, 34

API Application Programming Interface 66, 139, 156, 185

CVE Common Vulnerabilities and Exposures 19, 305

CWE Common Weakness Enumeration19, 25, 123 f., 133, 160, 174, 226, 257, 268 f., 272, 278, 283 ff.

DFA Data Flow Analysis52, 56, 62, 76, 210, 213 f., 217 ff., 223, 231, 233, 236 f., 239, 247, 256, 258, 304

DFD Data Flow Diagram 5, 29, 150, 301

DSL Domain-Specific Language11, 13, 19 ff., 23, 29, 45, 51, 85, 113, 147, 149 f., 154, 157–160, 162 ff., 166, 168–171, 173–177, 189, 192–196, 199, 263–267, 269–274, 277 f., 280–294, 299 ff., 303 ff., 307 f., 311 f., 314–318, 320

EMF Eclipse Modeling Framework . . . . . 15, 205 f., 211 f., 255 f.

GDPR General Data Protection Regulation 302

GQM Goal-Question-Metric . . . . . 181, 184, 187, 189, 193, 195 f., 282, 315

HLA High-level Architecture 48

IC Integration Condition 83 f., 91–104, 106–111, 114, 121 f., 141, 144 f., 174, 183–186, 194, 196, 200–203, 206–209, 230, 235, 237, 246 f., 250, 252 ff., 257–261, 314 ff., 322

IDE Integrated Development Environment 119, 303

JOANA Java Object-sensitive ANAlysis . 21, 26, 36 f., 39 f., 42, 55 f., 60, 63, 87 ff., 106, 112, 167, 170 f., 173 ff., 211 f., 233–237, 239 f., 242, 244, 247 f., 250 ff., 256, 258 f., 261

LLM Large Language Model 321, 323

MDSD Model Driven Software Development . . . . . 13 f., 17

OWASP Open Web Application Security Project 4, 25, 113, 115, 222, 226, 251, 257

PCM Palladio Component Model 17 f., 34, 47, 54, 62, 69, 76, 85, 114, 116 ff., 131, 138, 164, 168, 216, 221, 223, 235, 248, 278, 304 f.

RIFL RS^3 Information-Flow Specification Language 63, 303

SARIF Static Analysis Results Interchange Format 206, 303

SEFF Service-Effect Specification 18, 62, 76, 85

UML Unified Modeling Language . . . . . 17, 55, 62, 85, 305 f.

Part I.

Prolog

1. Introduction

This dissertation addresses the detection of architectural security vulnerabilities arising when a system's implementation does not comply with its architectural security design. Specifically, we[1] investigate how an architectural security analysis can be coupled with a static source code security analysis to detect architectural vulnerabilities that cannot be detected by applying either analysis in isolation.

This topic spans three aspects: *detecting vulnerabilities in software systems through security analysis*, *analysis coupling*, and the *connection between two system abstractions in software development* (i.e., the software architecture and the implementation). Each aspect itself is an active research area. However, their intersection — *coupling architectural analyses and source code analyses to detect architectural security vulnerabilities arising from non-compliance between architectural security design and the implementation* — has received limited attention in research. In this dissertation, we present our research results regarding this intersection. We provide freely available prototypes demonstrating the capabilities and challenges of our approaches.

The remainder of this chapter is structured as follows. Section 1.1 first motivates the relevance of security analysis in software systems and explores the challenges arising in architectural analysis when the implementation of a system does not comply with the architectural security design. We then identify problems arising for the coupling of architectural security analyses and static source code security analyses in Section 1.2. Based on these problems, we derive research questions in Section 1.3. Section 1.4 presents our contributions that address these problems and constitute our answers to the research questions. Finally, Section 1.5 outlines the remaining structure of this dissertation.

1.1. Motivation

Modern software systems often process or store sensitive information, such as financial data or health records. Most of these systems are connected to other systems or accessible from the outside (e.g., via the internet). If such systems handle sensitive information, they must meet security objectives like confidentiality and integrity to prevent unauthorized access or modification of data. Violations of these security objectives — for example due to

[1] While I am the sole author of this dissertation, I will use the term *we* because it is established practice in scientific writing. In addition, science is seldom done in isolation. Thus, I also want to give credit to all the people I have been fortunate to work with in the course of my research.

exploitation of *security vulnerabilities* in the system — pose risks for individuals and service providers alike. A *security vulnerability* is a weakness in the system (e.g., in its design or implementation) that an actor can exploit [319]. Exploitation of vulnerabilities can lead to attackers misusing personal data of the system users (e.g., for impersonation [88]). In the year 2024, a vulnerability in the system for managing the electronic health records in Germany was disclosed [106]. Exploiting this vulnerability would enable attackers to access all health records. Simultaneously, service providers face financial and reputational losses when vulnerabilities are exploited [48, 60, 88]. Vulnerabilities in safety-critical systems, like mobility systems, energy supply [186, 312], or healthcare [46], present particularly alarming cases for misuse.

Therefore, security vulnerabilities must be detected and fixed before they can be exploited. This task remains critical given the ever-rising number of vulnerabilities in software systems [333]. To detect such vulnerabilities, software engineers must review development artifacts, such as the system's security design and its implementation. However, these reviews are laborious and error-prone [77]. These challenges in security reviews first motivated research and development of static source code analyses that examine the implementation for various types of vulnerabilities without deploying an insecure system. Examples of addressed vulnerabilities are buffer overflows [86], insecure cryptographic API usage [176, 265, 366], and insecure information flows [20, 63, 285].

However, the later an issue in the system is fixed, the more cost-intensive the fix becomes [33]. Consequently, issues — in our case, *vulnerabilities* — should ideally be detected and fixed during the development phase in which they are introduced [299]. This practice is especially important for vulnerabilities introduced in the design phase, because design flaws cause approximately 50% of all vulnerabilities [81]. The Open Web Application Security Project (OWASP) Top Ten List of Security Risks [242] for 2025 further highlights this importance by ranking *Insecure Design* in sixth place. Therefore, design vulnerabilities must be detected and fixed before the system is implemented. This insight motivated the research and development of design-level security analyses such as UMLSec [154]. In this dissertation, we focus on *architectural security analyses*, a type of design-level security analysis, because architectural design decisions have a major influence on quality properties such as security [121].

Many architectural security analyses use input models that describe an architectural security design comprising the system architecture and security specifications (e.g., [12, 150, 154, 172, 300]). The system architecture contains the structure and optionally an abstract description of the behavior of the system. The security specification enriches the architectural model with security characteristics [299] relevant for the vulnerability detection. A security characteristic describes security-relevant properties of the system or its elements by a type and a set of values (e.g., the confidentiality levels of data with the values *confidential* and *non-confidential*).

Ideally, once a system is determined to be free of vulnerabilities by architectural analyses, it will be implemented according to the architectural security design. However, evolutionary changes can lead to a state where the implementation of a system fails to comply with the system design or its security specification [40, 159, 207, 256, 298]. Non-compliance may

occur because implementing a secure design requires software engineers to have expertise in security-related concepts. Software engineers may lack this expertise, although it is often assumed otherwise [284]. This insight is supported, for example, by studies showing that software engineers struggle with the correct application of cryptography [117, 226].

We distinguish between *design-related non-compliance* (sometimes called structural non-compliance [250]) and *specification-related non-compliance*, based on whether the implementation diverges from the system design — in our case the architecture — or the security specification. *Design-related non-compliance* occurs when the implementation does not realize the system structure or behavior described in the system design. Tuma et al. [337] present an example of design-related non-compliance, where encryption is omitted in the implementation despite being described in a Data Flow Diagram (DFD). In contrast, *specification-related non-compliance* occurs when the implementation does not realize security properties specified in the security specification. For example, consider a service parameter in the system architecture specified to contain only non-confidential data (using a security characteristic with the value *non-confidential*). If a static source code analysis now reveals an information flow of *confidential* data to this parameter in the implementation, specification-related non-compliance occurs. This non-compliance arises because the implementation results in the parameter containing *confidential* data rather than *non-confidential* data as specified in the security specification. The information detected here is in fact the values of a security characteristic that manifest in the implementation but are discovered by the source code analysis. We refer to such information as the *values resulting from the implementation*, which is further defined in Subsection 2.3.1 of this thesis.

This example also illustrates that specification-related non-compliance may arise when the implementation fails to comply with the system design but not vice versa. Consider the system architecture to describe that data should be encrypted (lowering its confidentiality level) before flowing to the parameter in our prior example. If this encryption is not implemented (design-related non-compliance), the confidentiality level of the data is not lowered as expected by the specification for the parameter (i.e., to contain only non-confidential data). Conversely, though, design-related non-compliance does not necessarily arise if specification-related non-compliance exists. An example is architectural security designs that replace a behavioral description in the system architecture by a more abstract security specification (e.g., [172]). Consequently, the implementation cannot be non-compliant to the system design, as the behavior is not considered. This is a crucial point, because it means that specification-related non-compliance may be difficult to detect by software engineers since it may be completely unrelated to the system architecture. This challenge is the reason why we focus on *specification-related non-compliance* in this thesis.

Non-compliance — especially specification-related non-compliance — also poses a significant challenge for architectural security analyses. Architectural analysis performed without information about the implementation must calculate vulnerabilities under the assumption that the implementation complies with the architectural security design.

Consequently, non-compliance may prevent architectural analysis from detecting vulnerabilities in the final system because the values specified in the design phase and the values resulting from the implementation diverge. As a result, engineers may incorrectly assume the system to be secure based on analysis results, while attackers can exploit vulnerabilities when the system is deployed and in operation. Such missed vulnerabilities are considered especially critical by developers, as they can lead to serious security incidents [14]. Similar to specification-related non-compliance, architectural vulnerabilities arising from a non-compliant implementation may be particularly hard to detect because their cause is unrelated to the artifacts investigated by the analysis. This is also why security must still be ensured in later stages of development, such as the implementation phase, even when performing architectural security analysis [300].

This dissertation addresses the challenges arising due to specification-related non-compliance by coupling model-based architectural security analyses with static source code security analyses to enrich the architectural analysis by the values of security characteristics that result from the implementation. The coupling allows to conduct the architectural analysis with values of security characteristics resulting from the implementation rather than based on values specified by a software engineer early in the design phase.

1.2. Problem Statement

As motivated in Section 1.1, specification-related non-compliance between the implementation of a system and its architectural security design poses a significant challenge for ensuring the security of software systems. Consequently, software engineers need to determine the impact a non-compliant implementation has on architectural security. As a first step, software engineers must be capable of *detecting* non-compliance between the architectural security design and the implementation. Related work addresses non-compliance detection (e.g., [145, 150, 159, 256, 295]) by employing source code analysis to evaluate whether the implementation realizes security-related information in the security design. However, these approaches merely report whether non-compliance exists, resulting in **Problem 1**.

Problem 1 - Incapability to Determine the Impact Of Non-Compliance on Design-Level Security: Non-compliance detection approaches in the security domain [145, 150, 159, 256, 295] leave determining the impact of non-compliance on design-level security to the software engineers. Without automated support, this task has to be performed in reviews manually which proves laborious and error-prone [77]. This task is particularly challenging because it requires relating design-level security and implementation-level security. However, this relation is not yet systematically captured [367]. Research on architectural erosion shows that software engineers performing the implementation may lack knowledge about the architectural design [188]. Consequently, we argue that determining the impact of non-compliance on design-level security is especially challenging as it requires knowledge about the system design — in our case the system architecture —, the implementation, and security.

Analysis composition allows combining the results of different analyses to generate new insights or to improve the results of an analysis [324]. Approaches in other domains of software engineering such as IObserve [120] or CIPM [213] demonstrate the feasibility of analysis composition to integrate implementation-level information into architectural analyses. Especially, Durán et al. [69] discuss the potential of analysis composition to improve systems or their specification. Based on these insights and ideas of non-compliance checking, we state following hypothesis: *Analysis coupling can enable the incorporation of information from static source code analysis into architectural security analysis, allowing to detect architectural vulnerabilities arising from a non-compliant implementation.*

However, coupling architectural analyses and static source code analyses faces the same challenge described in **Problem 1**: the gap between architectural security and the implementation-level security has to be bridged by relating information in the analyses in a meaningful way [71]. To the best of our knowledge, all non-compliance detection approaches (e.g., [145, 150, 159, 256, 334]) employ specific analyses, specific models of the system and investigate spectific security properties. Selecting specific analyses, models and properties has proven valid as those approaches are all scientific contributions. However, related work focuses on describing the relations between the specific analyses, models and properties rather than on a systematic process on how to uncover such relations. Consequently, there is currently no systematic process for determining the relation between architectural security and implementation-level security, exposed in the results of the architectural analysis and the source code analysis, respectively. In the context of our analysis coupling, this results in **Problem 2**.

Problem 2 - Lack of Systematic Approaches for Relating Implementation-Level Vulnerabilities to Architectural Vulnerabilities: No systematic approaches exist for relating vulnerabilities detected by source code analyses and vulnerabilities detected by architectural analyses to bridge the gap between architectural security and implementation-level security. However, these relations are a prerequisite for analysis coupling, especially because the compositionality of security properties — i.e., the relations between security properties investigated by analyses — is not yet well understood [71]. This lack results in repeated and unguided manual effort to determine meaningful relations between architectural analyses and source code analyses for new couplings. The significance of this task is highlighted by the creation of non-compliance detection approaches for specific analyses, models and properties (e.g., [95, 150, 159, 250, 334, 337]) being themselves scientific contributions. This problem is even more severe, because the relations to be determined must be valid on the syntactic level *and* semantic level to enable a valid analysis coupling [323, 335]. Furthermore, no approaches exist that detail feasible methods to facilitate information exchange between architectural analyses and source code analyses based on such relations.

Once the question of how to relate architectural security and implementation-level security is addressed by analysis coupling, appropriate analyses must be selected. This selection is affected by **Problem 3**.

Problem 3 - Lack of Approaches to Select Analyses for Analysis Coupling: Selecting analyses for the analysis coupling requires determining whether a source code analysis can be used to detect non-compliance that affects security vulnerabilities reported by the design-level analysis. This selection also requires determining whether the information available in both analyses suffices to establish the coupling. However, currently neither the relationship between vulnerabilities reported by source code analyses and security properties of the system is systematically captured [367] nor does an approach exist to investigate this relationship. Consequently, software engineers must manually select analyses for the coupling without systematic guidance. This selection is difficult because — as we will show in this thesis — even security experts are not fully aware of all relationships between design-level security descriptions — which are used to specify security properties of the system [174] — and findings of source code security analyses.

1.3. Research Objective

To address the problems identified in Section 1.2, we state the following research goal of this dissertation:

Research Goal: Our goal is enabling software engineers to investigate how non-compliance between the implementation and the architectural security design impacts architectural security. For this goal, we propose to couple architectural security analyses with static source code security analyses. To support this coupling, we aim to (1) specify properties that constitute an analysis coupling appropriate for coupling architectural security analyses and static source code security analyses, (2) provide approaches for relating implementation vulnerabilities detected with source code analyses to the architectural security investigated by an architectural analysis, and (3) provide an approach for selecting static source code analyses capable of detecting non-compliance that affects the architectural security investigated by an architectural analysis.

We derive four research questions to address this research goal. Analysis coupling requires understanding the properties that coupling approaches must exhibit for a given goal. For example, Talcott et al. [323] describe different information sources of the analyses that are combined (e.g., metamodels of the analyses, the analysis results or the analysis algorithms) or different approaches on how the information in the analyses is combined. The selection of these properties depends on requirements imposed by the analyses to be coupled to achieve the intended goal. We refer to the properties that coupling approaches for a given goal must exhibit as *coupling design*. Specifically, Talcott et al. [323] highlight that the identification of conditions influencing the instantiation of these properties remains an open challenge. This challenge leads us to ask **Research Question 1** to address **Problem 1**.

Research Question 1 - Suitable Properties of Coupling Approaches: What properties must coupling approaches exhibit to be suitable for the coupling of architectural analyses and source code analyses to investigate the effect of non-compliant implementations on architectural security?

Once the coupling design is defined, the information exchange between the analyses must be realized. This task requires bridging the gap between the information in the architectural analyses and source code analyses to be coupled. Approaches which bridge the gap between design models and the implementation for performing non-compliance analyses can be found in various related works [150, 159, 250, 295, 334, 337]. However, we did not find related work that systematically investigates how to relate vulnerabilities found by source code analyses to the architectural security investigated by architectural security analyses. The required relations must also be valid on the syntactic and semantic level; otherwise the coupling itself is invalid [323, 335]. Therefore, we ask **Research Question 2** to address **Problem 2**.

Research Question 2 - Relations to Bridge the Gap Between Architectural Security and Implementation-Level Security: What relations between information of source code analyses and architectural analyses can be used to enable bridging the gap between architectural security and implementation-level security to facilitate the coupling?

Research Question 1 and **Research Question 2** investigate the theoretical foundation for the coupling of architectural security analyses and static source code security analyses. However, the information exchange must be realized on a technical level. Therefore, we ask **Research Question 3** to address **Problem 1** and **Problem 2**.

Research Question 3 - Mechanisms to Enable Information Exchange: What are mechanisms appropriate to enable the information exchange between architectural analyses and source code analyses?

To determine whether two analyses can be coupled, it is necessary to select a source code analysis capable of detecting vulnerabilities that affect the architectural security investigated with the architectural analysis. **Research Question 2** supports the selection process only partially because the relations identified can be exploited only if a source code analysis with this capability is already selected. Thus, this research question only addresses part of **Problem 3**. The relation between security vulnerabilities reported by source code analyses and security properties is not yet systematically captured [367]. These security properties are commonly specified in design-level domain-specific languages, which are also used to define the input models for architectural analyses. Therefore, we can assume that no approach exists that allows relating architectural analyses with source code analyses that are capable of detecting vulnerabilities that affect architectural security involving such properties investigated with the architectural analysis. This capability is a prerequisite for selecting source code analyses suitable for the coupling. Consequently, we ask **Research Question 4** to address **Problem 3**.

Research Question 4 - Information Suitable for Selecting Source Analyses for the Coupling: Which information allows relating source code analyses and architectural analyses so that the source code analysis is capable of detecting non-compliance that affects architectural security investigated with the architectural analysis?

1.4. Contributions

This dissertation investigates the coupling of architectural analyses and static source code analyses to detect architectural vulnerabilities arising from a non-compliant implementation. In this context, we investigate which properties constitute coupling approaches appropriate for coupling architectural analyses and static source code analyses for our goal. We also investigate how to bridge the gap between both types of analyses and how

to determine which architectural analyses and source code analyses can be coupled for our goal. We focus on coupling architectural security analyses — a subclass of design-level analyses — because architectural design decisions have a major influence on quality properties such as security [121]. Our coupling is also described on static source code security analyses because they are widely used to detect vulnerabilities in implementations [49].

C1: Coupling Design. We provide a coupling design that defines properties suitable for coupling an architectural security analysis with a static source code security analysis to detect design-level vulnerabilities arising from a non-compliant implementation (coupling design). In **C1**, we define, amongst others, (1) the coupling goal, (2) the coupling form (what is combined — e.g., metamodels, results or algorithms), and a coupling process (how the analyses interact). For this definition, we consider challenges from coupling analyses typically applied in different stages of the software development lifecycle. Analysis coupling requires that the information in the analyses is semantically related [323]. Consequently, we investigate these semantic relations and their influence on the analysis coupling. This contribution presents our answers to ❓ **Research Question 1**.

C2: Coupling Approaches. Building on **C1**, we provide two coupling approaches that instantiate the coupling design for coupling architectural analyses with two different major types of source code analyses [49]. The first approach couples architectural analyses with specification-based source code analyses (also called property analyses [49]) to investigate architectural vulnerabilities that arise from information flows which are not compliant with the architectural security design. The second approach couples architectural analyses with pattern-search-based source code analyses (also called bug-finding analyses [49]) to investigate architectural vulnerabilities that arise from patterns indicating encryption vulnerabilities violating the architectural security design. These approaches exploit different types of semantic relationships between the analyses and use different methods to bridge the abstraction gap between the analyses. **C2** answers ❓ **Research Question 2** and ❓ **Research Question 3** by detailing relations and mechanisms used to bridge the abstraction gap between the analyses. Furthermore, **C2** contributes to ❓ **Research Question 4** because the relations elaborated to bridge the abstraction gap can also be used to determine whether two analyses can be coupled.

C3: Relating Security Domain Specific Languages and Static Source Code Security Analyses. We introduce *SecLan*, an approach for relating security Domain-Specific Languages (DSLs) and static source code security analyses. *SecLan* provides the *SecLan* conceptual model, a conceptual model capturing concepts and relations between concepts of security-focused DSLs [352] (security DSLs) and of static source code security analyses. We synthesized this model by reviewing 66 security DSLs and 33 static source code analyses. Security DSLs are important means to design secure systems that meet their security requirements [26] and are commonly used to describe input models for architectural analyses. Using the *SecLan* conceptual model, *SecLan* calculates relations indicating whether a source code analysis can be applied to detect non-compliance regarding security designs described by a security DSL. *SecLan* generalizes to diverse design-level analyses and is not restricted to architectural analyses. Because security DSLs are commonly used to describe input for architectural analyses, **C3** forms our answer to ❓ **Research Question 4**.

1.5. Outline

This dissertation consists of four parts: *Part I — Prolog*, *Part II — Contributions*, *Part III — Validation* and *Part IV — Epilog*.

The remainder of *Part I — Prolog* — in which we are currently in — describes foundations necessary to understand the concepts and approaches presented in Chapter 2 and provides a running example in Chapter 3. The running example is used throughout this thesis to illustrate scientific problems and the concepts in our contributions. The running example also illustrates the problem arising from non-compliance between architectural security designs and their implementation for architectural analyses on a specific example.

Part II — Contributions presents our three contributions of this dissertation. We start with the *coupling design* (**C1**) in Chapter 4, continue with the *coupling approaches* (**C2**) in Chapter 5, and conclude our contributions with our approach for *relating security domain-specific languages and static source code security analyses* (**C3**) in Chapter 6.

Part III — Validation presents the validation of our contributions. We perform a joint evaluation of **C1** and **C2** because the coupling design (**C1**) can only be evaluated by assessing our coupling approaches instantiating the design (**C2**). Furthermore, the evaluation of **C3** differs in the evaluation objectives and data collection methods applied from the evaluation of **C1** and **C2**. Thus, to provide a clear structure, Chapter 7 first presents the evaluation objectives and discusses the data collection methods appropriate for evaluating our contributions. The study design for evaluating **C1** and **C2** and the results obtained are then presented in Chapter 8. Finally, the study design for evaluating **C3** and the results obtained are presented in Chapter 9.

Part IV — Epilog finalizes the thesis. We start in Chapter 10 with a discussion of related work regarding our contributions. In Chapter 11 we then conclude this thesis with a summary of this dissertation and a discussion of benefits of our contributions. Chapter 11 also comprises future work to address limitations of our contributions and for advancing the field of connecting design-level security and implementation-level security.

2. Foundations

This chapter presents the foundations necessary to understand the concepts and approaches presented in this dissertation. This dissertation addresses the coupling of design-level analyses and source code analyses to detect the impact of a non-compliant implementation on design-level security. Consequently, our research involves the intersection of software security engineering, the relation between design-level security and implementation, and analysis coupling. We present established terminology and definitions in these areas where available and introduce new definitions or refine existing ones where needed. We highlight new definitions or the refinement of existing ones using horizontal bars above and below the respective text.

The remainder of this chapter is structured as follows. Our approaches address model-based analyses and use model-driven technologies to achieve the analysis coupling and to relate security DSLs and static source code security analyses. Therefore, we introduce the foundations of model-driven software development in Section 2.1. As stated in our contributions (see Section 1.4), we focus on coupling *architectural* security analyses with static source code security analyses. While we assume that the target audience is familiar with the basic concepts in software implementation — especially source code — we see the introduction of concepts in software architecture in Section 2.2 as beneficial. We then introduce foundations in the context of software security engineering in Section 2.3. Based on these concepts, we introduce the topic of security analyses in Section 2.4, where we also introduce types of analyses relevant in this thesis. Finally, we introduce the topic of analysis composition and its relations to analysis coupling in Section 2.5.

2.1. Model-Driven Software Development

In Model Driven Software Development (MDSD), models are the primary artifacts in the development process [35, 317]. For the approaches in this dissertation, we leverage concepts and technologies of MDSD. One of the core principles of MDSD is metamodeling which is concerned with defining the structure of models [234, 317]. Another principle is *Model Transformations*, which use instances of metamodels — i.e., models — to generate other models or source code [35, 317]. Valid model transformations must correctly consider the *syntax* and *semantics* of the involved model elements. Various techniques can be applied to query or manipulate these models regarding various information. In particular, we apply *Regular Path Expressions* in **C3** to search for paths in models represented as graphs. In the following, we introduce all these concepts in more detail.

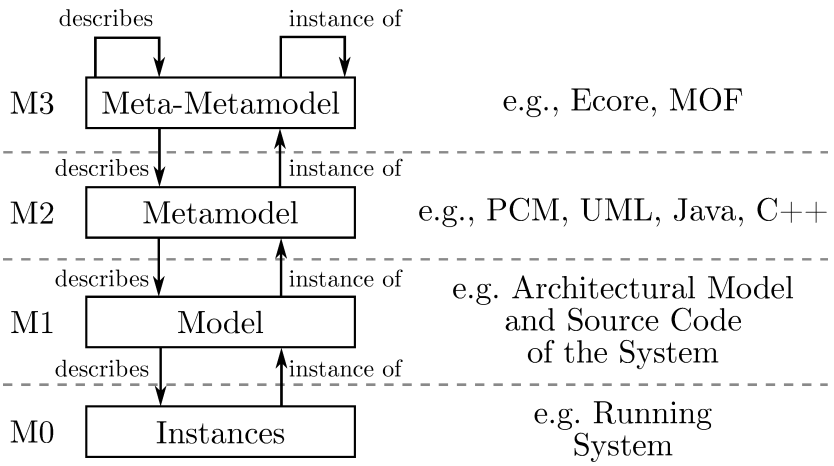


Figure 2.1.: Abstraction layers in the metamodel hierarchy (M0 to M3) with examples from this thesis for each level. Adapted from Stahl and Völter [317].

2.1.1. Metamodeling

Stachowiak [316] defines three key features of a model: A model is a *representation* of some natural or artificial entity (*mapping feature*) with a subset of the entities' attributes (*reduction feature*) for a specific purpose (*pragmatic feature*). Note that we also comprehend source code as *models* because it represents the system's structure and behavior (mapping feature), while commonly omitting details such as the hardware used and the system deployment (reduction feature) for the purpose of executing the system (pragmatic feature). This notion is supported by approaches like the *Java Model-Parser and Printer* [119], which represent Java source code as models and allow to manipulate these models and generate source code from them.

MDSD uses the concept of *metamodeling* to define the structure of models [234, 317]. To describe this structure, metamodels contain *metaclasses*, and directed relations between the *metaclasses* called *references* [234, 317]. With this definition, a metamodel describes the structure of models by defining the types of elements (i.e., metaclasses) that can be used in the models and the relations (i.e., references) between these elements. Furthermore, constraints for the model elements can be defined [234, 317].

To formally describe metamodels and models, we adopt the set-theoretic notation of Burger [42] and Klare et al. [167]. In this formalism, a metamodel X specifies the metaclasses $x_n \in X; n \geq 0$ and references $\langle x_i, x_j \rangle; i, j = 0 \ldots n$ between these metaclasses. A model instance ($\underline{X}$) conforming to a metamodel X contains instances of metaclasses $\underline{x} \in \underline{X}$ and instances of references $\langle \underline{x_i}, \underline{x_j} \rangle; i, j = 0 \ldots n$ between the instances of metaclasses. We define the function $hasType(\underline{x}, \underline{y}) : \underline{X} \times X \rightarrow \{true, false\}$ that queries for an instance $\underline{x}$ in a model $\underline{X}$ whether it is of the type (i.e., the metaclass) $y \in X$ in the metamodel X. We denote a set of metaclasses with uppercase letters (e.g., M) and the set of instances of these metaclasses with underlined uppercase letters (e.g., $\underline{M}$).

A metamodel itself can be described by another model, called *meta-metamodel* [234, 317]. A meta-metamodel could again be described by another model and so forth. However, in this thesis, we consider four layers of abstraction in the metamodel hierarchy (M0 to M3) as presented by Stahl and Völter [317]. In this hierarchy, we distinguish between the instance layer (M0), the model layer (M1), the metamodel layer (M2), and the meta-metamodel layer (M3). Every model at a lower layer is an instance of the model at the next higher layer. The meta-metamodels at layer M3 are self-describing; they define themselves and — in turn — are also instances of themselves [317]. We show this hierarchy in Figure 2.1 which also comprises examples from this thesis for each level. For example, a component-based architectural model describing a specific system (M1) is an instance of a metamodel (M2) describing the structure of architectural models. Similarly, the source code for a specific program (M1) is an instance of a programming language (M2) describing the structure of programs written in that programming language.

Prominent examples of meta-metamodels are the Meta Object Facility [234] or Ecore [320] used in the Eclipse Modeling Framework (EMF). For the implementations of our approaches in this dissertation, we use the EMF [320] — a popular framework for model-driven development. For more information about EMF, please refer to the EMF book [320].

2.1.2. Model Transformations

Model transformations translate one or more input models into one or more output models [35]. The transformations are described based on *transformation rules* that define how elements in the input models are translated to elements in the output models [317]. The transformation rules are described based on the *metamodels* of the input and output models [238, 317]. Based on this definition, we also comprehend manual translations between models as well as analyses as model transformations. We comprehend model-based analyses as model transformations, because they receive models as input (e.g., the source code) and provide their results again as models (see later in Section 2.4). We simplify the presentation of concepts in our approaches, we describe transformations as one-to-one transformations [35] (i.e., one input model is transformed into one output model). However, the transformations commonly use multiple input models to generate one or more output models [35].

The transformation rules are described based on *correspondences* between the elements of the input and output models. The correspondences define relations between elements of the respective models which are formulated on metamodel level [147, 297]. When transformation rules are applied, correspondences between elements in the input model and elements in the output model can be captured in a *correspondence model*, which is based on a *correspondence metamodel* [120, 147].

We denote a metaclass capturing a correspondence between metaclass $a \in A$ and metaclass $b \in B$ as $a \leftrightarrow b$. This notation can also be applied to the model level, where $\underline{a} \leftrightarrow \underline{b}$ denotes a correspondence between an instance $\underline{a}$ (of the metaclass a) and an instance $\underline{b}$ (of the metaclass b).

2.1.3. Syntax and Semantics of Languages and Models

A language or model is created based on its *syntax*. The *syntax Syn* of a model is a possibly infinite set of *syntactic elements* $syn \in Syn$ which convey information [115]. However, the syntax itself does not provide the meaning of the syntactic elements [115]. Consequently, Harel and Rumpe [115] state that syntactic elements must map to a *semantic domain Sem* to have a well-defined meaning. A semantic domain comprises *semantic elements* $sem \in Sem$ that describe concepts within a given universe of discourse [115]. Semantic domains are composable, i.e., they can comprise different semantic domains. Establishing a meaning for syntactic elements is performed by a *semantic mapping* $SM\colon Syn \rightarrow Sem$ [115]. This semantic mapping relates a syntactic element to a semantic element in a semantic domain.

2.1.4. Graph-Based Representation of Models and Regular Path Expressions

Models can be represented as directed graphs [74, 297]. We use the definition of Ebert and Bildhauer [74], where a graph consists of a set of typed nodes $n \in N$ and typed edges $e \in E$. An edge represents a directed connection between two nodes $n_x, n_y \in N$ with the signature $e\colon N \times N$ (denoted as $n_x \rightarrow n_y$). The type of the nodes and edges are defined through a type function $type\colon (N \cup E) \rightarrow T$, where each node and edge is mapped to a type $type_t \in T$. For brevity, we denote the node type by $n_x \in N$ by $n_x\colon type_n \in T$. We discuss the notation for the edge type in the following when we introduce regular path expressions.

In the context of metamodels and models, each instance of a metaclass in a model is described as a node with the type being the metaclass. Similarly, each reference between instances of metaclasses in a model is described as an edge between the corresponding nodes with the type of the edge being the type of the reference.

Regular Path Expressions can be used to define queries over such graph-based representations [74]. Starting at a given node n_1, the path expression $path$ is a regular expression over the edge types of a graph. The expression allows describing the direction over which an edge must be navigated to form the path: An edge type can be navigated *in the direction* of the edge (denoted by an arrow pointing to the right $\rightarrow$) or navigated *opposite to its direction* (denoted by an arrow pointing to the left $\leftarrow$). The type of the edge $type_e \in T$ is represented by annotating it with a label: $\xrightarrow{type_e}$. Edges can be defined to be navigated *exactly once* (denoted by $\xrightarrow{type_e}$ or $\xleftarrow{type_e}$) or *an arbitrary number of times* (denoted by $\xrightarrow{(type_e)^*}$ or $\xleftarrow{(type_e)^*}$). For the complete syntax and semantics of regular path expressions, see Ebert and Bildhauer [74].

Regular path expressions can be used as queries over the graph, returning all paths matching the specified expression. Equation 2.1 illustrates a regular path expression matching all paths from node a of type $type_a$ to node b with type $type_b$, where an edge of

type $type_x$ is navigated in the direction of the edge from a to an intermediate node and an edge of type $type_y$ is navigated opposite to its direction.

$$\begin{aligned} &path(a \xrightarrow{\text{type}_x} \xleftarrow{\text{type}_y} b) \\ &\text{with} \qquad a : type_a, b : type_b \in N; \quad type_x, type_y \in T \end{aligned} \tag{2.1}$$

2.2. Software Architecture

Software architectures describe fundamental concepts or properties of a software system in its environment [142]. An architecture description can comprise one or more architecture views, each addressing specific concerns of stakeholders [142]. Valipour et al. [342] highlights the role of the software architecture to describe the system's structure (e.g., the interacting parts the system is composed of), top level design decisions and the main pathways of interaction.

In MDSD, software architectures are commonly described by architectural models created with Architecture Description Languages (ADLs) [51]. These ADLs specify "the syntax and semantics intended for use in describing the architecture of an entity of interest" [142]. According to the ISO 42010 standard [142], ADLs are not only used for capturing the system structure but also for performing architectural analysis. An ADL can capture different architectural views, such as the structural view (e.g., components and connectors), the behavioral view (e.g., interactions between components), and the deployment view (e.g., mapping of software components to hardware nodes) [142, 275].

In this thesis, we focus on structural views and deployment views of software architectures, as they are relevant for our concepts. These views are provided, for example, by the Palladio Component Model (PCM) [275] or Unified Modeling Language (UML) [235]. In this thesis, we choose the PCM [275] as an example ADL to illustrate our concepts, because it is used in several security analyses (e.g., [111, 172, 300, 348]), comprises comprehensive architectural information and is adopted in the community [111]. However, our concepts are not limited to the PCM and can be applied to other ADLs that support similar architectural views.

The content of the PCM can be classified into three viewpoints [275] which we will discuss in the following.

> **Structural Viewpoint.** The structural viewpoint describes the system's structure. The PCM follows a component-based approach, where the system is composed of components that interact with each other through defined interfaces. For this purpose, the PCM comprises the *Component Repository Model* which contains the components and their interfaces. Each component can provide or require the services defined by an interface using respective provided and required roles. For example, components can represent an online shop and a logger. Logging services are represented as an interface which is provided by the logger component. In turn, the interface comprising the logging services is required by the online shop component to define the need for performing logging. The structural viewpoint of the PCM also includes

the *System Model* which describes how components are assembled to form a system. This model includes, amongst others, connectors between the provided and required roles of components.

Deployment Viewpoint. The deployment viewpoint describes the mapping of software components to hardware nodes and how these nodes are connected to each other. The PCM includes the *Resource Environment Model* which describes the hardware nodes and their connections, and the *Allocation Model* which describes how software components are allocated to hardware nodes.

Behavioral Viewpoint. The behavioral viewpoint describes the behavior of the system as well as the interaction of the user with it. The PCM allows specifying the internal behavior of components using a Service-Effect Specification (SEFF) for each provided service. The SEFF describes the control flow of the service, including calls to other services, loops, and branches. The PCM also includes the *Usage Model* which describes how users interact with the system.

With these viewpoints, the PCM allows capturing architectural information relevant for security analyses, such as the components and their interactions, the mapping of software components to hardware nodes, and the behavior of the system. Including its comprehensive tooling for analysis in various domains (e.g., performance, reliability, security), the PCM is a suitable example ADL for illustrating our concepts.

2.3. Secure Software Engineering

In this section, we introduce definitions and terminology used in secure software engineering. Security analyses are also part of secure software engineering. However, as these analyses are the core focus of this dissertation, we defer the definitions and terminology for them to Section 2.4.

We start with introducing general security terminology. We then continue with introducing the concept of security by design as secure software engineering requires considering security in every phase of software development. Finally, we address the topic of non-compliance between the system design and the implementation as part of secure software engineering.

2.3.1. Security Terminology

Software security is the capability of a software system to maintain *security objectives* [318] (also called *security properties* by Allen et al. [10] or the ISO-Standard 27002 [140]). Common *security objectives* include data *confidentiality*, data *integrity* and data or system availability [140, 318]. *Confidentiality* ensures protection of information from unauthorized entities. *Integrity* ensures that information cannot be modified or destroyed by unauthorized entities. *Availability* ensures that information is accessible to authorized

entities when needed. In addition to these basic *security objectives*, also other objectives are defined such as *non-repudiation*, *accountability*, *privacy*, or *authenticity* [140].

Security objectives can be violated by *threats* [139] which exploit *vulnerabilities* in the system. The NIST [31] defines a *vulnerability* as one or more weaknesses that can be accidentally triggered or intentionally exploited by a threat. *Weakness* refers to an undesired characteristic of a system's requirements, design, implementation or operation [31]. The Common Weakness Enumeration (CWE) database [327] provides a catalog of common types of software weaknesses. Similarly, the Common Vulnerabilities and Exposures (CVE) database [328] provides a catalog of known vulnerabilities in software systems.

Although standards like NIST [31] or ISO [139] provide clear distinctions between weaknesses and vulnerability, scientific literature about design-level analyses and source code analyses often use the term *vulnerability* for security issues found with security analyses (e.g., [12, 49, 66, 103]). Thus, we use the terms *weakness* and *vulnerability* in the context of analyses as follows. We use the term *weakness* to describe a security issue that *can be found* by an analysis. In contrast, we use the term *vulnerability* to describe an issue that *is reported by an analysis*. This definition includes a worst-case scenario, where every issue reported by an analysis is assumed to be exploitable. This aligns with the use of the term *vulnerability* for security issues existing in the system used in the terminology applied by the CVE database [328].

Chess and West [49] differentiate vulnerabilities into the classes *generic* and *context-specific*. *Generic* vulnerabilities *can* occur in almost every implementation (e.g., buffer overflows). In contrast, *context-specific defects* arise due to characteristics of a specific system. For example, insecure handling of credit card data can only arise in systems that process payment information (context-specific defect) while deadlocks — which violate the availability of the system — can occur in any system.

2.3.2. Security by Design

Engineering secure systems requires considering security throughout the whole software development process, especially during the system design [204]. This is also one reason why processes like the Microsoft Security Development Lifecycle [210] exist that prescribe planning and assessing security in every stage of system development and maintenance. Considering security already in the design phase of a system is also called *security by design*. One important aspect is planning and realizing *security features* which introduce mechanisms to ensure security. Designing software systems upfront regarding a certain property or the planning of security features can be realized with *Domain-Specific Languages (DSLs)*.

2.3.2.1. Security Features

Security-critical parts of the system are secured by *security features* (sometimes also called *security mechanisms*, e.g., by Stallings [318]) to ensure security objectives are met. These

features realize mechanisms in a software system, that protect assets by preventing or mitigating threats [204] (e.g., input sanitization, encryption or access control). Thus, security features are important means to achieve secure software systems [204, 318]. However, Chess and West [49] point out that vulnerabilities in the software systems arise not only from issues in security features (e.g, weak cryptography), but also from issues in non-security related parts of the implementation (e.g., buffer overflows through which sensitive information can be accessed).

In this dissertation, we focus on two security features: *data encryption* and *information flow control*. Data encryption transforms data into a format that is unreadable for unauthorized entities [176, 318]. Commonly, encryption is used to ensure data confidentiality during data storage or communication [318]. In the implementation, software engineers can use cryptographic APIs [176, 366] to apply encryption. However, the correct use of these APIs is challenging and misuses can lead to vulnerabilities [366].

Information flow control is the capability of a system to control how information flows in the system [63]. For example, information flow control can ensure that confidential data does not flow to non-confidential system parts. Information flows are established *explicitly* through transmitting data (e.g., by assignment statements) or *implicitly* by the observable behavior of the system (e.g., through conditional branches) [20, 63, 285]. Information flow control is especially relevant as information flows can result in violating the security objectives confidentiality and integrity [285]. Security features such as access control or data encryption [285] are commonly used to control the flow of information. However, as Sabelfeld and Myers [285] point out, once data is decrypted, it must be ensured that the receiver of the data respects the security objectives. Consequently, information flow control does not solely rely on security features (e.g., cryptography or access control) but also on other approaches like security policies and analyses. Consequently, we classify information flow control itself as a security feature despite it involves application of security features. We introduce the *analysis* of information flows in more detail in Subsubsection 2.4.2.4.

2.3.2.2. Security Domain-Specific Languages

A Domain-Specific Language (DSL) is a language that allows to focus on a dedicated domain and has a limited expressiveness [352]. *Security DSLs* allow the specification of, and the reasoning about, design-time security concerns [26]. Amongst others, security DSLs also allow the planning of security features in the design of a system.

We found that security DSLs can be commonly structured into the *system design* and the *security specification* (e.g., found in [12, 89, 150, 154, 172, 192, 300, 338, 348]). The *system design* describes the structure and behavior of a system on a specific level of abstraction (e.g., system-related business processes [288], data flow diagrams [304, 338] or component diagrams [154, 172, 300, 348]).

The *security specification* extends this system design with security-related concepts in the *security specification* by providing security-related properties to elements describing the system structure or behavior. We refer to these security-related properties as *security*

characteristics [299], while other terminology like security label [318] or security class [63] is also used. Please note that security characteristics are not only described in security DSLs but can also be specified in general-purpose programming languages such as Java (e.g., [19, 98, 105, 255]).

A security characteristic consists of a type and a set of values [299]. We simplify in this article by interpreting the values of a security characteristic as instances of its metaclass. Security characteristics describe a wide variety of security-relevant properties: Hahner et al. [112] use a security characteristic with the values *EU* and *Non-EU* to describe the location of resources (European Union or not). Kramer et al. [172] use security characteristics to describe protection mechanisms against tampering with resources by values such as *Seal* or *Special Screws*. UMLSec [154] describes the protection of communication links, e.g, with values *encrypted* or *LAN*. Johannes Geismann and Bastian Haverkamp and Eric Bodden [150] use security characteristics to specify that a component performs *input sanitization*. In information flow control, *security characteristics of data* are often used to describe access control policies based on values representing roles or confidentiality levels. For instance, the CANUKUS example of Bell [25] or the TravelPlanner system [159, 211] uses roles to specify conditions on when an actor in the system may access data. These roles are represented by basic values (e.g., *UK* (United Kingdom), *US* (United States) in the CANUKUS example) or as their combination (e.g, *UKUS*). We interpret basic values as combined values with only one sub-value. Confidentiality levels commonly comprise non-combinable values forming a strict order (e.g., *Confidential* (*high*) and *Non-Confidential* (*low*) used to design secure data flows [63, 338]). Another example are confidentiality levels with the values *Top Secret*, *Secret*, *Confidential*, and *Unclassified* used by the military [291].

The specification of security characteristics is attached to system representations by various mechanisms [123] (e.g., stereotypes in UMLSec [154, 300], annotation metamodels [34], Java annotations in *Java Object-sensitive ANAlysis (JOANA)* [98] or textual representations in FlowDroid [19]). We refer to all of these mechanisms with the term *annotation*, because annotations are a common mechanism to provide information to models representing the system [123]. Other specification mechanisms can exist such as using a dedicated parameter in a design-level element (e.g., applied by Tuma et al. [338]). However, such mechanisms can be represented using annotations for increased modularity [123].

2.3.3. System Design and Implementation Compliance

In this thesis, we consider compliance between the system design specified in design-level models and the implementation of the system. Compliance in this context describes whether the implementation realizes the system and its security-related aspects as specified in design-level models. Existing approaches [1, 150, 159, 256, 298, 337, 363] allow checking the implementation for compliance with the design regarding specific security aspects. These compliance checks often involve configuring source code analyses (e.g., [150, 159, 337, 363]) or utilizing patterns that violate design-time security aspects (e.g.,[145, 252]).

Tuma et al. [337] define an implementation as compliant if a source code analysis does not report a vulnerability when analyzing behavioral information derived from the design. This definition of compliance is focused on the system description. However, we also found that the implementation may not comply with the security specification while the actual system design itself does not capture the responsible information. Thus, we distinguish two distinct types of non-compliance as introduced in Section 1.1: *design-related non-compliance* and *specification-related non-compliance*. *Design-related non-compliance* occurs, when the system design explicitly describes elements which are not realized in the implementation. For example, design-models presented by Seifermann et al. [300] or Tuma et al. [338], describe an element representing the encryption of data. *Design-related non-compliance* occurs if this encryption is not realized in the implementation. In contrast, *Specification-related non-compliance* occurs when the design-level model specifies security characteristics for design-level elements which are not fulfilled in the implementation. We illustrate this type of non-compliance in Section 1.1. In the given example, non-compliance arises because the implementation contains an information flow of confidential data to a parameter which is specified to contain only non-confidential data. Specifically, this information flow is not captured in the design-level model. For *Specification-related non-compliance*, we specialize the definition of compliance provided by Tuma et al. [337] in Definition 1.

Definition 1 ***(Non-Compliance and Non-Compliant Values of Security Characteristics between Implementation and Design-Level Specifications)***

Given a design-level model which specifies security characteristics for design-level elements, and a source code analysis that provides information in its output to derive the values resulting for implementation elements. We denote the implementation as non-compliant *with a security characteristic specified for a design-level element if the values of the security characteristics derived for an implementation element cannot be mapped to the values specified for a corresponding design-level element.*

This definition of non-compliance can be used to investigate whether the implementation complies with the security characteristics specified in a design-level model. We define an implementation as non-compliant (*non-compliant implementation*) if at least one non-compliant value of a security characteristic is found according to Definition 1. In Definition 1, the static source code analysis inspects the source code and provides information that allows deriving the values of security characteristics that are the result of the behavior realized in the implementation. This is why, we denote the values uncovered by a source code analysis as the *values resulting from the implementation*, because they manifest in the implementation but are uncovered by the source code analysis. Therefore, the values of security characteristics can manifest either by specification in the input models of architectural analyses or as a result of a static source code analysis.

2.4. Security Analyses

This section introduces definitions and terminology regarding *security analyses*. In this dissertation, we focus on the coupling of *model-based analyses*. Based on this definition, we introduce *source code analyses* and *architectural analyses* as specific types of model-based analyses.

2.4.1. Model-Based Analysis

A *model-based analysis* evaluates *input models* M_{in} with an *analysis technique* T to provide an answer to a given *question* Q in their *output models* M_{out} [324].

The syntax of the input models and output models can be described by metamodels or grammars. We focus in this thesis on analyses that use metamodels and models [317] because we build upon related work on metamodel evolution for model-based analyses [123]. Similarly to comprehending source code as models (see Subsection 2.1.1), we also perceive *static source code analyses* as *model-based analyses*, because they receive the implementation as input model and provide an output model.

The *input* of design-level analyses is commonly described using security DSLs (see Subsection 2.3.2). The input of static source code analyses is the source code of the implementation. Certain static source code analyses also require security specifications, as we discuss in Subsubsection 2.4.2.2.

In security analyses, the *question* to be answered typically concerns whether the input model contains a specific security vulnerability. Examples for questions are whether the system uses weak cryptographic algorithms [366], or whether information flows to untrusted sinks exist [198, 285]. To answer these questions, various *analysis techniques* are possible, such as label propagation or formal verification.

The *output* of a model-based analysis is an abstraction of the input models [323]. Output models can provide information in various forms, such as numeric values, boolean statements (e.g., whether the system is secure or not), or counter-examples describing why a vulnerability exists [323]. For simplicity, we define a model-based analysis to use only one input model and to provide one output model. We are aware that the input and output of an analysis can be distributed over several models. However, this simplification does not limit the applicability of our concepts and approaches presented in this thesis.

Security vulnerabilities can be detected manually or by applying (semi-) automated tools [49]. Such tools can answer one or more *questions* by investigating the input models. However, the terminology used in literature in this domain is not consistent. Literature describe software performing the investigation of questions with terms like *analyzer* (e.g., [36, 228, 305]) or *analysis tool* (e.g., [12, 324]). There is also related work [30, 190, 191] that use these terms interchangeably. Additionally, for individual investigations of questions, different terms are used in literature. While in the design domain, the term *security analysis*

is often used (e.g., [12, 113, 150, 299]), the term *security check* (or the task *security checking*) is also sometimes used in the source code domain (e.g., [190, 191]).

To provide consistent terminology in this thesis, we use the following terms: We use the term *analysis* for the investigation of a single question *if we are not interested in the relation regarding a specific analysis tool.* We use the terms *analyzer* and *security check* if the tool performing the analysis is relevant and connections to the questions it answers are established.

2.4.2. Static Source Code Security Analyses

Static source code security analyses use the source code of a system as input and provide detected vulnerabilities as output [49]. In accordance with model-based analyses, we perceive the source code analysis as a mapping $\mathcal{A}_C$ between the input model of the source code analysis C and the output model of the source code analysis R_C as described in Equation 2.2.

$$\mathcal{A}_C \colon C \rightarrow R_C \tag{2.2}$$

Several analyses exist that transform the source code into an intermediate representation, (e.g., into abstract syntax trees or a control flow graphs) to perform the analysis [49].

For source code analyses, various classifications exist in literature. We discuss these classifications in detail in Subsubsection 2.4.2.1, while also introducing new categories for classifying source code analyses relevant for this thesis. We put special attention on two classes of source code analyses described by Chess and West [49] that are relevant for this thesis: *property checking analyses* and *bug-finding analyses.* While we will introduce the general classification of Chess and West [49] involving these classes in Subsubsection 2.4.2.1, we will discuss both of them in more detail in Subsubsection 2.4.2.2 and Subsubsection 2.4.2.3 respectively. Furthermore, we introduce information flow analyses as a specific type of source code analyses in Subsubsection 2.4.2.4, because they are relevant for our contributions and are commonly used to detect security vulnerabilities in implementations [19, 20, 98].

2.4.2.1. Classifications of Source Code Analyses

There are different classifications of security analyses in literature. We provide a non-exhaustive overview of these classifications. However, as we will see, these classifications focus on the analysis techniques employed or how developers can use the analyzers rather than the security vulnerabilities they detect. Thus, we also introduce a classification based on the types of security vulnerabilities and the variety an analyzer can detect.

> *Automatization.* Chess and West [49] and Talcott et al. [324] categorize analyses based on their level of automation into manual analysis, semi-automated analysis, or fully automated analysis.

Analysis Purpose. Chess and West [49] categorize analyzers into different *categories*. We comprehend the term *category* as too broad and refer to it as *purpose*. The classes which Chess and West [49] define are, amongst others, type checking analyses, program understanding, bug-finding analyses, property checking analyses or security review analyses.

Syntactic/Semantic Analysis Lipp et al. [190] differentiate *syntactic* and *semantic* source code analyses. *Syntactic* analyses search for patterns in the source code, which is similar to the bug-finding analyses by Chess and West [49]. *Semantic* analyses leverage additional information for analysis, such as control flow or data flow by lifting the source code to a more abstract representation.

Programming Language. McGraw [204] categorizes analyses based on the programming language they support, such as Java or C.

Format of Implementation. AlBreiki and Mahmoud [9] differentiate analyses based on the format of the implementation they are analyzing: *source code*, *byte code* or *binary*.

Purpose of Output. Nachtigall et al. [225] differentiate tools by the purpose of the output, for example, *Understandable Warning Messages*, *Fix support*, or *User Feedback*.

Interface. Nachtigall et al. [225] also categorizes analyzers based on the interfaces they provide into *Command-Line Tools*, *Standalone*, i.e., with a separate graphical user interface, *IDE tools*, or *Tools with multiple interfaces*.

The prior classifications for source code analyses focus on the analysis techniques employed, the purpose of the analyses or how developers can use the analyzers rather than the specific security vulnerabilities they detect. Scientific literature about analyses (e.g., [138, 366]) and other sources like the OWASP Source Code Analysis Tools List [241], commonly classify analyses according to the types of vulnerabilities they are capable to detect. For this purpose, they mainly use the CWE as a reference, which categorizes weakness by its influence on high-level security objectives, such as *confidentiality*, *integrity*, *availability*, *access control*, or *non-repudiation*. However, these objectives are very broad, which makes them hard to use as description for the capabilities of source code analyses. Therefore, taking into account the above classifications and considering further literature, we identified the following classification of analyses based on the types of security vulnerabilities they can detect:

1. *API – Security API Usage:* This category includes analyses with the capability to identify the insecure usage of security-related APIs (e.g., cryptographic APIs). Analyses such as *CogniCrypt* [176] focus on cryptographic APIs by checking for weak encryption algorithms or creating a secret key with hard-coded constants. This category is mentioned by Zhang et al. [366] and Kulenovic and Donko [178].
2. *IF – Information Flow:* This category includes analyses with the capability to identify unallowed information flows in the source code. Analyses like *JOANA* [98] examine information flow — that is, data flow and control flow that may carry information implicitly [20, 135, 341] — to identify violations of a given security policy (i.e.,

confidential information must not flow to non-confidential locations). Taint analyses propagate taint labels along data flows but often only consider explicit flows [20]. This class is mentioned by Kulenovic and Donko [178]. We exclude analyses from this class that use information flow analysis to detect other kinds of issues. We will discuss information flow analyses in more detail in Subsubsection 2.4.2.4.

3. *D – Dependency:* This category includes analyses with the capability to identify the usage of dependencies (e.g., libraries or frameworks) that are known to be vulnerable in the specified version used. Analyses such as *OWASP Dependency Check* [240] or Snyk [310] examine dependencies (e.g., as specified in build systems such as Maven) to determine whether used libraries contain known but unfixed vulnerabilities. This class is mentioned in the survey of Imtiaz et al. [138] and used by the approach of Kirschner et al. [165].

4. *VCP – Vulnerable Coding Practices:* This category includes analyses with the capability to identify coding practices known to be insecure. Examples of such insecure coding practices are code prone to buffer overflows [55], null pointer dereference [148], or using inherently dangerous functions (e.g., *FlawFinder* [354]). In contrast to *Security API Usage*, these coding practices result in security vulnerabilities without direct association with security-related functionality, such as encryption or access control.

5. *P – Permissions:* This category includes analyses with the capability to identify problems related to permission management. Examples of such problems are missing or problematic permission checks (e.g., *PaX* [365]) or more permissions being claimed than necessary (e.g., Tang et al. [325]). This category also relates to access control vulnerabilities described by Kulenovic and Donko [178].

We classify *analyzers* based on the categories of vulnerabilities they can detect into *single-purpose analyzers* and *multi-purpose analyzers*. As discussed before, an *analyzer* can provide one or multiple security checks. If an analyzer provides security checks for *multiple types of vulnerabilities*, we call it a *multi-purpose analyzer*. If an analyzer provides security checks for *one type of vulnerability*, we call it a *single-purpose analyzer*. Note that this classification focuses on types of vulnerabilities provided by the analyzer and not the number of the security checks. We classify CodeQL [100, 214] as *multi-purpose analyzer* because it provides security checks detecting, amongst others, (1) the usage of the weak encryption algorithm *DES* in the Java Cryptographic API (type *API*) and (2) information flows to public outputs (category *IF*). We classify *JOANA* [98] as a *single-purpose analyzer* because it provides only one security check for detecting unallowed information flows (category *IF*). We also classify CogniCrypt [176] as a *single-purpose analyzer* because it provides multiple security checks for the category *API*.

2.4.2.2. Property Checking Analysis

Chess and West [49] describe the purpose of a *property checking analysis* to investigate whether a given source code realizes a specification. Specifications commonly abstract the implementation and describe security-relevant properties of the system. Source code

verification tools like KeY [7] as well as information flow analyses tools like JOANA [98] or FlowDroid [19] fall in this category. Here, an example property to investigate is that confidential data must not flow to untrusted sinks (e.g., log files).

Thus, analyses in this class use the source code and specification as input and commonly provide counter-examples as output [49]. These specifications are described specifically for the system to be analyzed. To highlight the usage of specifications in the analysis input, we call these analyses *specification-based* in the remainder of this thesis. Note that also architectural analyses can be assigned to this class as they commonly use specifications and architectural models as input (see Subsection 2.4.3) to determine whether a given property is fulfilled.

For fully automated analyses, also a *security policy* has to be provided to the analysis. A security policy defines when a property is fulfilled or violated [294]. Consequently, the question to be answered by a specification-based analysis is whether the source code violates the specification given a security policy. To the best of our knowledge, there is no definition for security policies considering security characteristics. However, we found several model-based analyses that utilize some form of security policy using security characteristics in source code analyses (e.g., [98, 159, 285, 338]) or architectural analysis (e.g., [112, 172, 300, 338]). Therefore, we provide Definition 2 for security policies considering security characteristics.

Definition 2 ***(Security Policy)***

A security policy *defines when the analysis should report a vulnerability based on the specification of allowed or unallowed relations between values of security characteristics.*

2.4.2.3. Bug-finding Security Analyses

Bug-finding analyses [49, 136] use only the source code as input to search for bug patterns that are known to potentially cause vulnerabilities. These bug patterns are commonly described in a *pattern language* specific to the analysis. Some analyzers allow defining custom patterns (e.g., *CodeQL* [100, 214] or PMD [260]), while others provide only a fixed set of patterns (e.g., *FlawFinder* [354] or *SonarQube* [313]). In the remainder of this thesis we call *bug-finding analyses* also *pattern-search-based analyses* to emphasize the focus on what is used to detect potential vulnerabilities (i.e., the patterns). Thus, the terminology of *pattern-search-based analyses* aligns with the terminology of *specification-based analyses* which puts the focus on the use of specifications to detect potential vulnerabilities. As no specification is used, the analysis can only check for *generic vulnerabilities* (see Subsection 2.3.1).

If the analysis finds a pattern, it commonly provides the location in the source code and a corresponding pattern identifier in its output model. The patterns found can indicate errors in the usage of security features. A common pattern, detected by various analyses [18, 52, 260, 310, 313], is the usage of a *weak encryption algorithm* in the Java Cryptographic API (category API in Subsubsection 2.4.2.1). However, also patterns can indicate issues that are

not related to a certain security feature. An example for such patterns are buffer overflows which are not specific to a certain security feature, but can result in vulnerabilities. The capability of detecting these different kinds of patterns aligns with the remark of Chess and West [49], stating that vulnerabilities may arise in the implementation of security features but also in source code that is not deemed security related by developers. The patterns together with additional metadata are captured in *rules*.

2.4.2.4. Information Flow Analysis

Information flow analysis for source code investigates whether an information flow violates the specification (e.g. [19, 80, 98, 100, 224]). We use the term information flow analysis to refer to both information flow analysis and data flow analysis, as both investigate the data flow while the former also considers the control flow [20, 135, 341].

These analyses are commonly *specification-based*; that is, they use specifications to determine whether an unallowed information flow exists between implementation elements of the analyzed program. Some specifications define the role of implementation elements like parameters or fields in an information flow as the origin of a flow (*Source*) or a potential destination of flow (*Sink*), to inspect whether information flows between them exist (e.g., [19, 98, 100]). Some analyses use specifications that define a security characteristic of data for implementation elements to inspect whether unallowed information flows exist between them due to the contained information (e.g., [98, 224]). The values specified can vary to define different kinds of properties, such as confidentiality levels or roles (see Subsubsection 2.3.2.2).

Information flow analyses commonly use a security *lattice* as *security policy* [291] to describe allowed or unallowed information flows between values of a security characteristic of data [285, 291]. The lattices are commonly described transitively [198]; thus, if a flow is allowed from value A to value B and from value B to value C, then the flow is also allowed from value A to value C. The lattices and the allowed flows between values of a security characteristic of data are specific to the system and the property to be analyzed [285, 291]. Still, we found examples in literature of commonly used security characteristics and their interpretation in their corresponding lattices.

The first example relevant of commonly used security characteristics relevant in this thesis are roles (see Subsubsection 2.3.2.2). When using security characteristics of data representing roles, the *lattice* allows describing whether the combination of roles is interpreted *conjunctive* or *disjunctive*. In the *disjunctive lattice* (e.g., in the TravelPlanner [211]) an information flow is only allowed from a source system element to a sink system element if the latter is specified to contain *at least one* of the sub-values of those values specified for the source element. This lattice implies that an actor requires at least one of the comprised basic roles to be allowed to access the annotated information. In such a lattice, combined roles are less confidential than their sub-roles, because more actors in the system are allowed to access them. In *conjunctive lattices*, an information flow is only allowed from a source system element to a sink system element if the latter is specified to contain *all* of the

sub-values of those values specified for the source element. An example of a *conjunctive lattice* for confidentiality is described by Tuma et al. [338], where an information flow from the values *Alice* or *Bob* is allowed to sinks which are specified with the value *AliceBob*, but not vice versa. As Tuma et al. [338] describe, *Alice* is less confidential than *AliceBob*.

Another example for a commonly used security characteristic is indicating the confidentiality of data (see Subsubsection 2.3.2.2). Commonly, the lattice represents a strict order between the values of the security characteristic of data, which allows describing the confidentiality of data. In this lattice, the most confidential value is the one that is not allowed to flow to any other value, while the least confidential value is the one that is allowed to flow to all other values.

Based on the information flows found and the security policy, the analysis can determine whether a vulnerability exists. For determining whether a vulnerability exists, the analysis determines the information flows between the implementation elements and checks whether the flow between the values is allowed by the security policy. If the flow is not allowed, the analysis reports a vulnerability by providing a counter-example, which describes the information flow that violates the security policy. Such a counter-example commonly includes information such as the source and sink of the flow and — optionally — the values of the security characteristic of data.

2.4.3. Architectural Security Analysis

Architectural analyses use an architectural model of the system and a specification of security characteristics as input, and provide detected vulnerabilities as output. Thus, we classify them as *specification-based* analyses (see Subsubsection 2.4.2.2). In the remainder of this thesis, we perceive architectural analyses $\mathcal{A}_A$ as a mapping between the input model of the architectural analysis A and the output model of the architectural analysis R_A as described in Equation 2.3.

$$\mathcal{A}_A \colon A \rightarrow R_A \tag{2.3}$$

The input models of architectural analyses are *architectural models* described with ADLs — a specific type of security DSL (see Section 2.2 and Subsubsection 2.3.2.2). The architectural models used by the analyses commonly comprise the system structure (e.g., components and connectors; see Section 2.2). In addition, various architectural analyses also consider behavior specifications (e.g., [96, 111, 154, 300, 348]). The security-related information in the specification contains various aspects like attacker information [172, 338, 348], application of encryption [154, 338], access control information [192] or deployment information [12, 112, 300].

Architectural analyses — and design-time analyses in general — can be used for two purposes in relation to the implementation: First, they detect vulnerabilities in the architectural design of properties also available in the implementation. For example, the data flow analysis of Tuma et al. [338] detects data flow vulnerabilities in DFDs of the system. The benefit of using analyses for this purpose is that they can be applied before the

implementation is available [299]. Secondly, architectural analyses can be used to detect vulnerabilities that cannot be detected by source code analyses because this detection is only possible by considering information commonly not available in the implementation. Examples of such information include deployment information [12, 112, 172, 300].

2.5. Composition of Analyses

In model-based analysis composition, analysis models and analysis tools are the primary artifacts that are composed or decomposed [323]. The composition of the models is realized on the syntactic, semantic, and metamodel level [323]. In this thesis, we focus on the question *what* is connected from two analyses and *how* the analyses interact.

We see analysis coupling closely tied to analysis composition as defined in Definition 3.

Definition 3 ***(Analysis Coupling)***

Analysis Coupling *approaches provide the means to connect two analyses to achieve a specific purpose using a specific type of composition.*

A specific analysis coupling is realized based on a given *coupling scenario* as defined in Definition 4. The coupling scenarios in this dissertation focus on specification-related non-compliance according to Definition 1.

Definition 4 ***(Coupling Scenario)***

A coupling scenario *comprises an architectural analysis, a source code analysis, and a security characteristic under investigation. The security characteristic under investigation is the subject of the coupling. The values of this security characteristic specified for architectural elements in the input model of the architectural analysis are changed in the coupling if non-compliance is detected with the source code analysis.*

Talcott et al. [323] define two central properties of analysis coupling: The *coupling form* describes the information sources used in the coupling (e.g., metamodels, results or algorithms). The *composition type* describes how the coupling is achieved, i.e., how the information provided by the analyses is combined to obtain new results. We discuss these properties in detail in the following sections. Because the selection of the *coupling form* and *composition type* depends on the intended *coupling goal*, we introduce the term *coupling configuration* in Definition 5.

Definition 5 ***(Coupling Configuration)***

A coupling configuration *defines the* coupling goal*, the* coupling form *and the* composition type*. The coupling form and composition type must be selected to address properties specific to the analyses to be coupled for achieving a specific goal.*

2.5.1. Coupling Form

Talcott et al. [323] identify three *forms* of composition: *model composition*, *analysis composition* and *result composition*. These forms of composition differ in *what* is composed in the analyses. We refer to these forms of composition by the terms *white-box composition*, *grey-box composition* and *black-box composition* respectively, as these terms better express the level of visibility into the internals of the analyses. In the following, we discuss these forms in accordance with the definitions presented by Talcott et al. [323].

In *white-box* composition, the metamodels of analyses are combined, e.g., by definition of new metamodels that combine the metamodels of other analyses. The internals of the metamodels and analyses are open for modification on an arbitrary level of detail.

In *grey-box* composition, the steps of two or more analysis algorithms are coordinated to compute a result. The internals of the analyses are partially visible through interfaces to allow for this coordination. This coordination requires the introduction of new functionality into the analyses for synchronization and communication. Frameworks such as OPAL [125] or the High-level Architecture [57] provide such functionality so that information calculated by one analysis can be sent to another analysis.

In *black-box* composition, the input and output models of the analyses are combined by performing transformations between them. Additionally, the execution of the individual analyses is orchestrated. The syntactic and semantic relations between the elements of the input and output (meta)models must be clear so that no knowledge about the analysis algorithms is required [323]. The analysis techniques and analysis tools remain unmodified. There is no interaction between the analyses required during their execution.

2.5.2. Composition Type

Talcott et al. [323] identify different *types* of analysis composition that describe *how* information in the analyses is combined. Examples for these types are *simple result composition*, *model decomposition and result composition* and *sequential composition*. We will describe these types in the following in accordance with the definitions presented by Talcott et al. [323]. For a full list, refer to Talcott et al. [323].

Simple result composition combines the results of two or more analyses to provide a new result. In this composition type, the analyses are executed independently of each other and a new result is computed based on the individual results.

Model decomposition and result composition decomposes the input model of an analysis into multiple sub-models. These sub-models are then analyzed independently with different analyses. The results of these analyses are then combined to provide a new result.

In *sequential composition*, information in the output model of one analysis is made available for another analysis. Talcott et al. [323] refer to the process of making the information available as *parameterization*. The *parameterization* can be performed with different

techniques. The parameterized analyses may require a model which can be extracted from the output of the first analysis (e.g., parameters for the calculation performed).

3. Running Example

This chapter introduces the running example used throughout this thesis to illustrate our contributions. As described in Section 1.4, **C2** comprises two coupling approaches to address different security characteristics used in the architectural analyses and two major types of source code analyses. The running example is intended to illustrate both coupling approaches. Consequently, our running example comprises four parts:

- A system comprising an architectural design and implementation.
- An architectural analysis.
- A specification-based source code analysis investigating information flows in the system.
- A pattern-search-based source code analysis searching for patterns in the implementation that indicate invalidation of data encryption.

For a system and implementation to be useable for our running example, it has to satisfy two requirements. First, the architectural design has to comprise security characteristics that are relevant for both coupling approaches. Secondly, the implementation has to be non-compliant regarding these security characteristics so that the problems arising from non-compliance can be illustrated. Consequently, the implementation has to comprise both information flow vulnerabilities and encryption-related vulnerabilities that can be detected by the respective source code analyses. To the best of our knowledge, no system exists in related work that satisfies these requirements. Therefore, we extend the architectural design of the online shop system used by Hahner et al. [112] and provide an implementation so that both artifacts meet our requirements. We assume that every component and the corresponding class are developed by different teams with different levels of expertise. This introduces the possibility, for example, that the implementation of one component contains a vulnerability while the others do not.

The remainder of this chapter is structured as follows. In Section 3.1, we describe the architectural analysis, the architectural design and specification to which the analysis is applied and the obtained analysis results. Section 3.2 then describes the applied source code analyses, the implementation and specification to which the analyses are applied and the obtained analysis results. Based on this scenario, we illustrate the problems arising from a non-compliant implementation when applying the architectural analysis in isolation to the source code analysis in Section 3.3.

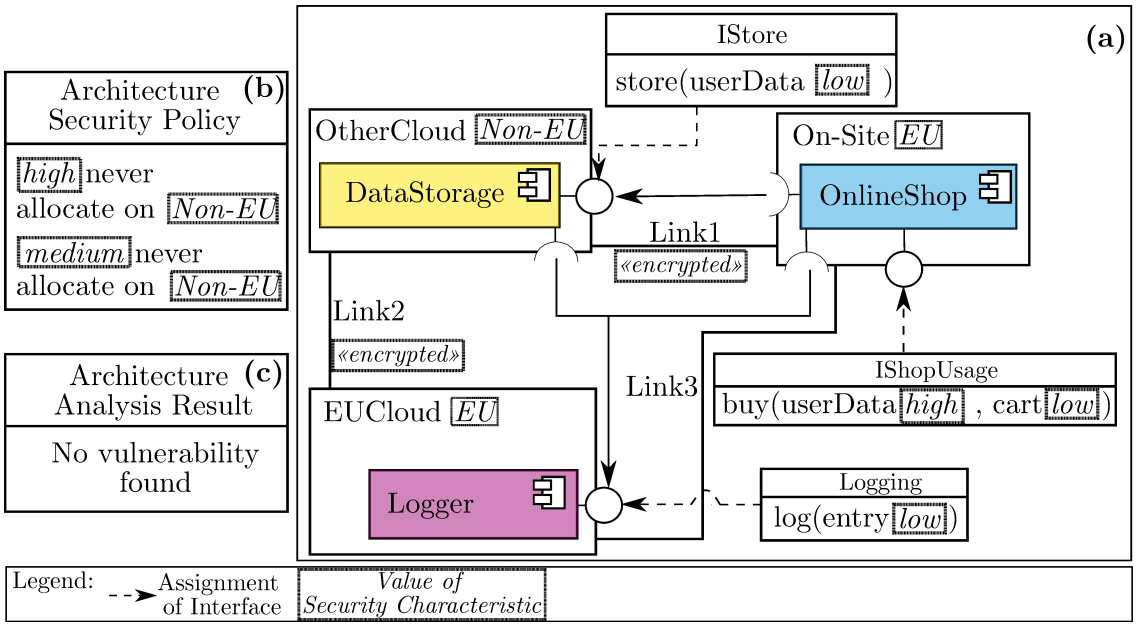


Figure 3.1.: Combined diagrams showing (a) parts of the architectural input model of the analysis, (b) the security policy used by the architectural analysis and (c) the output of the architectural analysis. Adapted from Hahner et al. [112]. Parameter types and return types are omitted for better readability.

3.1. Performing the Architectural Analysis

We describe an architectural analysis based on capabilities of the analysis of Seifermann et al. [300] and the *Secure Links* analysis of UMLSec [154]. This analysis reports architectural security vulnerabilities if (1) components handling confidential data are deployed in insecure locations, or (2) data is transmitted unencrypted over communication links connecting resources deployed in different locations.

We present the architectural design and its security specification which is used for the analysis. Then, we present the architectural analysis results.

3.1.1. Architectural Design

Partial model **(a)** in Figure 3.1 illustrates the architectural design of the online shop system in our running example. We describe the architecture of the online shop system using the PCM [275], an ADL for component-based architectures (see Section 2.2). We now discuss the architectural design regarding (1) the components and their services, (2) the deployment of the components on resources, and (3) the communication links between the resources.

Components and Services The system comprises three components *OnlineShop*, *DataStorage* and *Logger*. In our example scenario, the *OnlineShop* component represents the online shop application. The *DataStorage* component represents a cloud-based data storage

application used to store user data. The *Logger* component represents a logging service used to log events occurring in the system. For these purposes, the components provide and require the services as follows: The *OnlineShop* provides the service *buy(userData, cart)* to enable users to buy items. The parameter *userData* is used to provide details about the user and the parameter *cart* is used to represent the items a user wants to buy. The *OnlineShop* requires the service *store(userData)* which is provided by the *DataStorage* component. The *store* service stores the received *userData*. Both, the *OnlineShop* and the *DataStorage* components require the service *log(entry)* provided by the *Logger* component to log information. The parameter *entry* contains information to be logged.

Deployment The system comprises three resources *On-Site*, *OtherCloud* and *EUCloud* on which the components are deployed. First, the *OnlineShop* component is deployed on the *On-Site* resource. Secondly, the *DataStorage* component is deployed on the *OtherCloud* resource. Lastly, the *Logger* is deployed on the *EUCloud* resource.

Communication Links We employ three communication links between the resources: First, the communication link *Link1* connects the resources *On-Site* and *OtherCloud*, Secondly, the communication link *Link2* connects the resources *OtherCloud* and *EUCloud*, and Thirdly, the communication link *Link3* connects the resources *On-Site* and *EUCloud*

3.1.2. Architectural Security Specification

The architectural analysis uses three security characteristics: *Confidentiality Level*, *Location*, and *Encryption* to perform the analysis. We now describe for each security characteristic its purpose and the values used in our running example. Partial model **(a)** in Figure 3.1 illustrates the security characteristics annotated to the architectural elements.

Location *Location* is annotated to resources to describe the location they are deployed in. The values used in our running example are *EU* and *Non-EU* as described by Hahner et al. [112]. The resources *On-Site* and *EUCloud* are annotated with the *Location EU*. The resource *OtherCloud* is annotated with the *Location Non-EU*.

Encryption *Encryption* is annotated to communication links to describe whether data communicated over the link is encrypted or not. This information is represented by either annotating the communication link with «*encrypted*» or not (boolean value). Encryption is also specified in related work [154, 172].

The communication links *Link1* and *Link2* are annotated with «*encrypted*» because the communication between components deployed in the *EU* with components deployed outside the EU (i.e., value *Non-EU*) should only communicate data encrypted.

Confidentiality Level The *Confidentiality Level* is annotated to parameters of services to describe the expected confidentiality of data they contain. The values used in our running example are *high* (highly confidential), *medium* (confidential) and *low* (non-confidential). Similar values are used in confidentiality policies in related work [63, 112, 159, 338] making them relevant for our running example.

The parameter *userData* of the *buy* service is annotated with the *Confidentiality Level high* as it contains highly sensitive user information. The parameter *cart* of the buy service is annotated with the *Confidentiality Level low* because it does not allow deriving any sensitive information in our running example. The parameter *entry* of the *log* service and the parameter *userData* of the *store* service are both annotated with the *Confidentiality Level low*. This specification assumes that only non-confidential information is logged and that the *store* service only receives *userData* without sensitive information, because the *DataStorage* is deployed outside the EU.

3.1.3. Architectural Analysis Results

The architectural analysis investigates the security of the system based on the annotated security characteristics and the security policy (partial model **(b)** in Figure 3.1). The security policy is adapted from the one described by Seifermann et al. [300] and states that confidential data (i.e., values *high* and *medium*) must not be stored on resources located outside the EU (i.e., value *Non-EU*). Based on the annotated security characteristics and the security policy no vulnerabilities are detected by the architectural analysis (illustrated in partial model **(c)** in Figure 3.1).

The *DataStorage* is the only component deployed on a resource outside the EU (i.e., annotated with the value *Non-EU*). However, this component is expected to process only non-confidential data — received through the parameter *userData* of the *store* service annotated with the confidentiality level *low* — which is allowed according to the security policy. The other components are deployed on resources located in the EU which are allowed to handle data with all confidentiality levels. Therefore, no architectural vulnerability is reported regarding insecure deployment of components.

The communication links *Link1* and *Link2* connect resources with different values of *location* (i.e., *EUCloud* and *On-Site* annotated with *EU* and *OtherCloud* annotated with *Non-EU*). Both links are annotated with «*encrypted*». Thus, the architectural analysis also does not report an architectural vulnerability for these communication links.

3.2. Performing the Source Code Analyses

We describe the application of two source code analyses to the implementation of the online shop system. We use the specification-based information flow analysis *JOANA* [98] to investigate information flows in the system. *JOANA* is already used in related work [159] to analyze the compliance between design descriptions and the implementation. We also

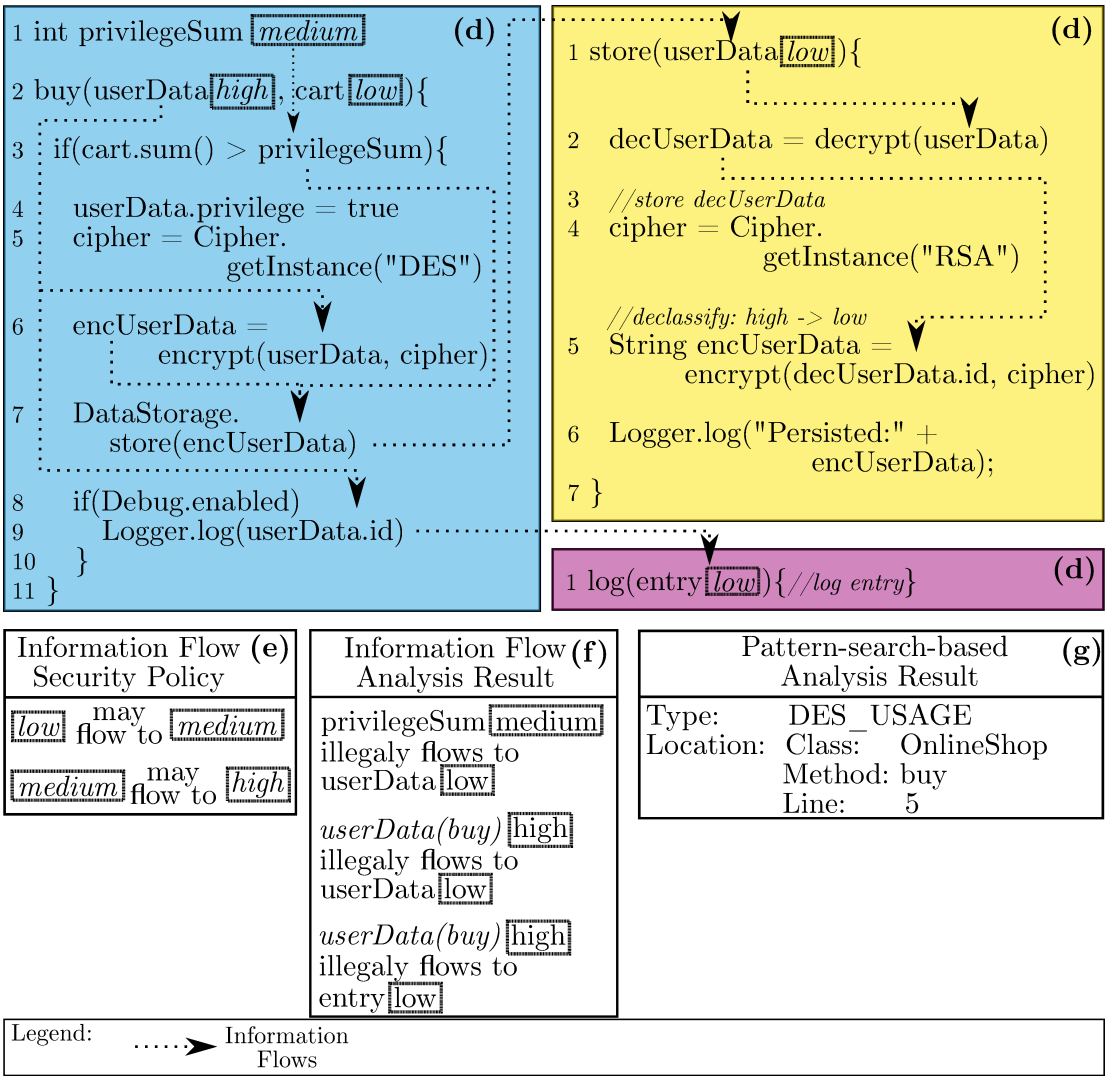


Figure 3.2.: Combined diagrams showing (d) parts of the implementations of the components from the architectural model, (e) the applied information flow policy, (f) the source code analysis results of the information flow analysis, and (g) pattern-search-based source code analysis. The colored areas indicate the implementation of classes corresponding to components with the same color in Figure 3.1. Parameter types and return types are omitted for better readability.

use the analysis *Find Security Bugs* [18] to detect patterns that indicate the erroneous usage of cryptographic APIs. *Find Security Bugs* is listed by literature [366] to detect such patterns.

We first describe the implementation of the architectural design in Subsection 3.1.1. Then, we describe the specifications for the implementation required by *JOANA* in Subsection 3.2.2. Finally, we describe the results of both source code analyses when applied to the implementation in Subsection 3.2.3.

All models used as input for the source code analyses and the results calculated by them are shown in Figure 3.2.

3.2.1. Implementation

As illustrated in the partial models **(d)** in Figure 3.2, the implementation includes the three classes *OnlineShop*, *DataStorage* and *Logger*. These classes map to the correspondingly named and colored components illustrated in partial model **(a)** in Figure 3.1. We discuss the implementation according to the classes that are implemented.

Implementation: Logger The *Logger* class provides the method *log* which takes a string *entry* as input and logs it in a database. We omit details about the logging process in the partial models **(d)** in Figure 3.2 for better readability and because the implementation logic is not relevant for our running example.

Implementation: Online Shop The *OnlineShop* class includes the method *buy* which implements the *buy* service of the online shop, an *encrypt* method for encryption of data and a field *privilegeSum*. We omit the *encrypt* method's implementation details in the partial models **(d)** in Figure 3.2 for better readability. The *encrypt* method encapsulates the encryption of *userData* and the conversion into a *String* representation. For this purpose, we define the *encrypt* method to receive the *userData* and a *Cipher* specifying the algorithm to be applied for the encryption. *userData* is then encrypted and returned as a *String* representation of the encrypted data.

With the *encrypt* method and the field *privilegeSum*, the *buy* method implements the following behavior: If the value of the items in the cart in the *buy* method is higher than the *privilegeSum*, the *userData* is modified by setting the *privilege* field to *true* and storing it in the *DataStorage*. This behavior is realized by the *if* statement which compares the value of the items in the cart (*cart.sum()*) to the value of *privilegeSum*. If the condition evaluates to true, the fields of *userData* are encrypted with the *encrypt* method and then sent to the *DataStorage*. This encryption is performed to protect the sensitive information in *userData* during transmission over communication link *Link1*, which is annotated with the security characteristic *encrypted* in the architectural model. For this purpose, the implementation generates the cipher using the algorithm *DES* (Cipher.getCipher("DES")) to encrypt the data using the Java cryptographic API. The *encrypt* method provides a *String* representation of the encrypted *userData* as output.

After the encryption, the provided String *encUserData* is sent to the *DataStorage* by calling the *store* method. If the debug mode is enabled (*if* statement with condition *Debug.enabled*), the *OnlineShop* logs the event by sending the *id* assigned to the userData (*userData.id*) by calling the *log* method of the *Logger*.

Implementation: DataStorage The *DataStorage* class implements the *store* service with the *store* method, an *encrypt* method to encrypt *userData* and a *decrypt* method to decrypt *userData*. We omit the *encrypt* method's implementation and the *decrypt* method's implementation in the partial models **(d)** in Figure 3.2 for better readability. The *encrypt* method is implemented analog to the *encrypt* method in the *OnlineShop* class; it receives

the *userData* and a *Cipher* specifying the algorithm to be applied for the encryption and returns a *String* representation of the encrypted data. The *decrypt* method takes the String representing the encrypted *userData* and provides the decrypted *userData* as output.

With this functionality, the *store* method is implemented by receiving the encrypted *userData* as input, decrypting it and storing it in a database. After storing the *userData*, the *DataStorage* logs the storage process using the user *id* contained in *userData*(*userData.id*). For this logging, the *DataStorage* encrypts the *id* conforming to the architectural specification where communication link *Link2* is annotated with the security characteristic *encrypted* to protect the data during transmission. However, in contrast to the *OnlineShop*, the *DataStorage* encrypts the *userData* with the secure *AES* encryption algorithm. The obtained string representation of the encrypted *id* is then sent to the *Logger* by calling the *log* method.

3.2.2. Specification for Information Flow Analysis

For illustration, we describe a simplified version of *JOANA* [98], which is based on the original capabilities of the analysis. In our simplified version, *JOANA* analyzes information flows between implementation elements annotated with *Security Levels* (security characteristics of data). These annotations are provided for *parameters* and *fields*. In this running example, we define *Security Level* to comprise the values *high*, *medium* and *low* analogously to the *confidentiality level* used in the architectural analysis. As shown in our partial models **(d)** in Figure 3.2, we annotate all parameters in the implementation with values corresponding to the values of the *confidentiality level* specified for corresponding parameter in the architectural model. For example, we annotate the parameter *userData* of the *buy* method with the value *high* as the corresponding parameter in the architectural model is annotated with the *confidentiality level high*. Similarly, the parameters *userData* and *entry* in the methods *store* and *log* are annotated with the value *low* respectively.

JOANA supports the specification of *declassification* of information, i.e., the change of the value of *Security Level* assigned to data to other values by the implementation without resulting in a vulnerability. This specification is provided for methods in the implementation by annotating them with the *declassify* annotation. Encryption is a mechanism commonly used for declassification. This annotation is necessary because *JOANA* cannot derive (1) whether a method declassifies information and (2) which value of *Security Level* the information is declassified to. We do not annotate the application of the *encrypt* method in the *buy* method of the *OnlineShop* class with declassification because it uses an encryption algorithm (DES) which does not provide sufficient security. In contrast, we specify *declassify* for the *encrypt* method in the *DataStorage* class because it sufficiently declassifies the data due to the usage of the secure encryption algorithm (AES).

3.2.3. Source Code Analysis Results

JOANA and *Find Security Bugs* detect vulnerabilities in the implementation described in Subsection 3.2.1.

Vulnerabilities Reported by JOANA *JOANA* uses the implementation and specification described before, and the security policy illustrated in partial model **(e)** in Figure 3.2. This security policy states that information of the confidentiality level *low* is only allowed to flow to implementation elements annotated with the confidentiality level *medium*. Conversely, information of the confidentiality level *medium* is only allowed to flow to implementation elements annotated with the confidentiality level *high*. These relations in information flow policies can be assumed to be transitive (see Subsubsection 2.4.2.4), so that information annotated with the confidentiality level *low* is also allowed to flow to elements annotated with the confidentiality level *high*. All other flows are forbidden and result in vulnerabilities.

Based on this information, *JOANA* reports three information flows that violate the security policy (illustrated in partial model **(f)** in Figure 3.2). The first information flow leaks highly confidential (*high*) information from the parameter *userData* of the *buy* method to *userData* of the *store* method which is expected to only contain non-confidential (*low*) data. This information flow arises because the *encrypt* method does not sufficiently declassify the *userData* as it uses an insecure encryption algorithm (DES).

The second information flow leaks confidential (*medium*) information from the field *privilegeSum* to *userData* of the *store* method which is expected to only contain non-confidential (*low*) data. This information flow arises because storing this data depends on the *medium* classified *privilegeSum* which introduces an implicit information flow (see Subsubsection 2.3.2.1). If the value of the *cart* (*cart.value()*) is higher than the value of *privilegeSum*, the *userData* is modified by setting the *privilege* field to *true* and storing it in the *DataStorage*. If an attacker can see the execution of *store* and is aware of the non-confidential *cart*, they can infer information about the value of *privilegeSum*.

The third information flow leaks highly confidential (*high*) information from the parameter *userData* in the *buy* method to the *entry* parameter of the *log* method. This flow arises because the *userData.id* is logged if the debug mode is enabled. Controlling the logging of sensitive data based on a debug flag is a combination of two known weaknesses: logging sensitive information [331] and a potential active debug mode [330].

Vulnerabilities Reported By *Find Security Bugs* *Find Security Bugs* uses the implementation to search for patterns indicating the erroneous usage of cryptographic APIs. *Find Security Bugs* reports the usage of the weak cryptographic algorithm *DES* in the *encrypt* method for the implementation presented before (illustrated in partial model **(g)** in Figure 3.2). The analysis reports an identifier of the pattern (*DES_USAGE*), the class, the method the pattern occurs in, and the line of code where the pattern is found. Please note that we

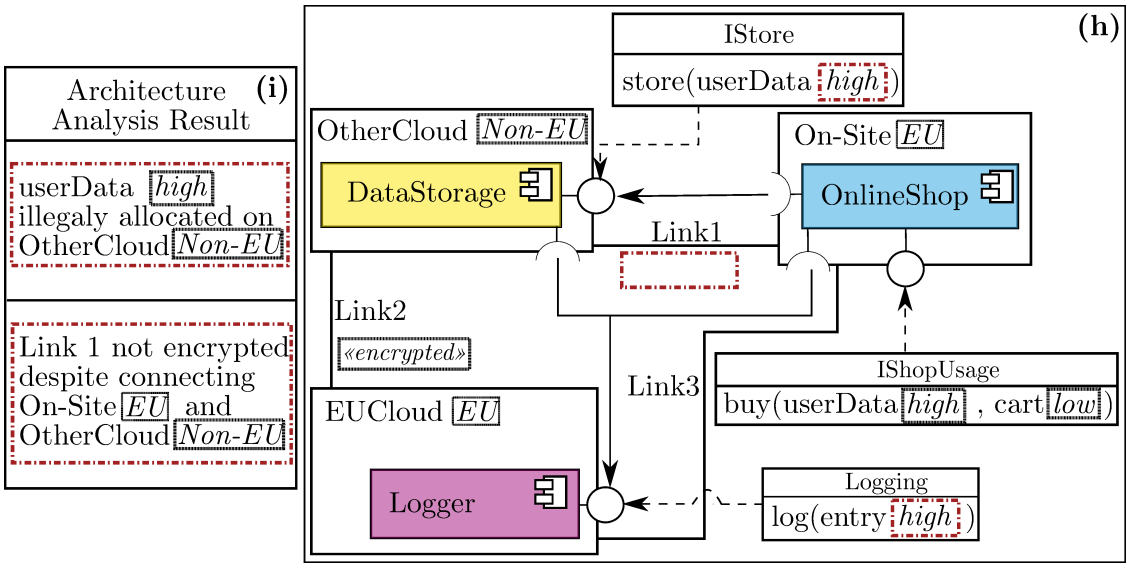


Figure 3.3.: Combined diagrams showing (h) the input model of the architectural analysis considering the values resulting from the implementation and (i) the architectural analysis result when considering the values resulting from the implementation. Red boxes indicate changes in the architectural design and the architectural analysis result.

abstract the syntax of the result provided by *Find Security Bugs* in Figure 3.2 for better readability.

3.3. Illustrating the Problem of Non-Compliant Implementations for Architectural Analysis

We now illustrate the problem in our running example arising from applying the architectural analysis in isolation of the source code analysis. For this purpose, Figure 3.3 shows the architectural model and the architectural analysis result when considering the values resulting from the implementation.

In our running example, values of the security characteristics *Confidentiality Level* and *Encryption* resulting from the implementation are non-compliant with values specified in the architectural model. Partial model **(h)** in Figure 3.3 illustrates the architectural model annotated with the non-compliant values resulting from the implementation. The parameter *userData* of the *store* method and the parameter *entry* of the *log* method actually contain high confidential (*high*) information due to the implementation rather than the non-confidential (*low*) information assumed in the design phase. Additionally, the communication link *Link1* is not sufficiently encrypted (not annotated with «*encrypted*») because the communicated data is not sufficiently secured due to encryption using the weak cryptographic algorithm *DES*.

Using these non-compliant values, the architectural analysis would report two vulnerabilities: one vulnerability regarding the deployment of *OtherCloud* and the second vulnerability regarding communicating data over *Link 1* (illustrated in partial model **(i)** in Figure 3.3).

In the following, we describe how these non-compliant values arise from the implementation based on the results of the source code analyses described in Section 3.2 and how these values would lead to the reporting of vulnerabilities by the architectural analysis.

The values resulting from the implementation are uncovered by the results of the source code analysis. The first two information flows reported by *JOANA* describe that information is leaked from *userData* in the *buy* method (classified with the value *high*) and *privilegeSum* in the *OnlineShop* class (classified with the value *medium*) to the *userData* in the *store* method (classified with the value *low*). Through these two flows, the parameter *userData* of the *store* method contains information classified with the *Security Level medium* or *high*. Both values do not comply with the value *low* specified in the architectural model. The third information flow leaks information from the *userData* in the *buy* method (classified with the value *high*) to the *entry* parameter of the *log* method (classified with the value *low*). This information flow results in the parameter *entry* to contain information classified with *high*, which does not comply with the value *low* specified in the architectural model.

Find Security Bugs detects the usage of the weak cryptographic algorithm *DES*, which is considered to be insecure [21, 22]. Consequently, the implementation does not comply with the architectural specification in which the communication link *Link1* is annotated «*encrypted*».

When considering these values, the architectural analysis *would* report two architectural vulnerabilities as illustrated in partial model **(i)** in Figure 3.3. The first vulnerability would be reported for the parameter *userData* of the *store* method used in the *DataStorage* component because now information with the *confidentiality level high* is processed on the *OtherCloud* resource with the *location Non-EU*. The second vulnerability would be reported because the communication link *Link1* connects resources with different *Location* values (i.e., *EU* and *Non-EU*) but is not annotated with «*encrypted*».

Part II.

Contributions

4. Designing the Coupling of Architectural Analyses and Static Source Code Analyses

Literature: This chapter is based on the following (co-) authored publications: [IEEE TSE 2025], [IEEE ICSA-C 2025]

Analysis coupling requires selecting a *coupling form* and *composition type* suitable for coupling the given types of analyses to reach a specific goal [323]. In the context of this thesis, selecting these properties requires considering challenges arising from the coupling of analyses that use different system abstractions and that are typically applied in different stages of the software development lifecycle. Therefore, we define a *coupling configuration* (see Definition 5) in Section 4.1 that is suitable for coupling architectural security analyses and static source code security analyses to investigate the effect of non-compliant implementations on architectural security.

Coupling analyses is a complex task which requires knowledge about the analyses to be coupled and knowledge about the relations between their information [324]. This task is especially challenging because the compositionality of security properties — in our case captured by the architectural analysis and source code analysis — is currently only limitedly researched [70]. The complexity of this task is highlighted because isolated security analysis is an individual topic in science (e.g., [12, 19, 98, 112, 154, 172, 224, 300, 338, 348]) and industry (e.g., tools like CodeQL [100, 214], Snyk [310], SonarQube [313]). The complexity of this task is also shown by related work which presents combinations of security DSLs and source code analyses for specific (meta)models or analyses (e.g., [150, 159, 256, 334]). A common approach to handle complexity in development tasks is to separate different responsibilities and competencies by assigning them to dedicated developer roles. For example, Reussner et al. [275] propose roles in the development of component-based architectures, such as *Software Architect, Deployment Expert or Component Developer*. Similarly, Koch [169] proposes roles for the composition of individual model-based analyses from analysis components, such as *Analysis Architect* and *Analysis Component Developer*. Consequently, as a first step to address the complexity in analysis coupling, we extend the roles proposed by Koch [169] by responsibilities and competencies for the development of coupled analyses in Section 4.2.

Determining valid relations between information in analysis coupling does not only require considering the syntax used in the analyses, but also the semantics [323, 335]. To address

this challenge, Section 4.3 discusses how the semantics of the information in the analyses influence the analysis coupling.

The *coupling configuration* selected must be realized in a *coupling process* that defines how the coupling is achieved. Consequently, we define a *coupling process* in Section 4.4, that outlines the necessary process steps and the information involved to achieve the coupling goal. However, defining a single coupling approach for all security properties and analyses is not feasible because of different types of relations arising from the information available in specific types of analyses. This difference must be addressed by dedicated methods to bridge the gap between the analyses. Thus, our coupling design — especially the coupling process — has to be *instantiated* by specific coupling approaches to address specific types of analyses and their information. We conclude this chapter with a discussion of limitations of the proposed coupling design in Section 4.5, and by providing a summary of the presented information in Section 4.6.

4.1. Coupling Configuration

In this section, we discuss the *coupling configuration* suitable for coupling architectural security analyses and static source code security analyses to detect architectural vulnerabilities arising from a non-compliant implementation. This discussion first includes defining the *coupling goal* (Subsection 4.1.1). For this coupling goal, the suitability of specific *coupling forms* and *composition types* are discussed in Subsection 4.1.2 and Subsection 4.1.3 respectively.

4.1.1. Defining the Coupling Goal

Our coupling goal is the detection of architectural vulnerabilities that originate from *specification-based non-compliance* (see Subsection 2.3.3) between the architectural specification and the implementation. These architectural vulnerabilities are especially relevant as they cannot be detected by the architectural analysis performed in isolation of the source code. Achieving this goal includes bridging the gap between vulnerabilities reported by source code analyses and vulnerabilities reported by architectural analyses.

For achieving this goal, we aim to design a coupling which only considers structural elements of the architectural design (e.g., components, services, resources, communication links) and a specification abstracting the behavior. Examples for such specifications are annotations to communication links to specify that data transmitted over the link must be encrypted (e.g., [154, 172]) or annotations specifying the expected confidentiality level of data contained in the parameter. We do not exclude architectural analyses from our coupling that use behavior descriptions (e.g., [158, 300, 348]) if they also provide structural information. However, we do not use any elements of behavior descriptions in architectural analyses for our coupling.

Our coupling does not use information in behavior description because architectural analyses either use no behavioral descriptions (e.g., [12, 172]) or more abstract behavioral descriptions than the implementation (e.g., [300, 338]). For example, Seifermann et al. [300] or Tuma et al. [338] describe the usage of encryption as a single action and omit details like the applied encryption algorithm. In the implementation, these actions have to be refined into more detailed actions, such as key management or selection of cryptographic algorithms, which can themselves be involved in vulnerabilities that violate the architectural specification (e.g., if weak encryption algorithms are selected or credentials are hard-coded).

4.1.2. Selecting the Coupling Form

The coupling form defines what is combined to achieve the coupling goal (e.g., metamodels, results or algorithms). In the coupling of architectural analyses and static source code analyses, the coupling faces the challenges arising from (1) analyses using different system abstractions and (2) the architectural analysis which is still required to be applicable in isolation of the source code analysis.

The capability of the architectural analysis to be applicable in isolation is necessary because architectural vulnerabilities should be detected and fixed before implementing the architectural design to avoid rising costs and efforts of late fixes [300]. This necessity arises either if the implementation does not exist (i.e., when developing a new system) or it is not yet evolved to the evolved architectural design (i.e., in software evolution). When the coupling does not maintain the capability to be applicable in isolation, two versions of the architectural analyses are necessary: (1) one version that can be performed in isolation of the source code analysis (i.e., without information from the implementation), and (2) one version which is used as coupled analysis.

To avoid the need for maintaining two versions of the architectural analysis, we select a black-box coupling. In the black-box coupling, the information exchange is realized by transformations between the input and output models of the analyses and by orchestrating the execution of the individual analyses [323]. Thus, this type of coupling does not require new analyses tooling or adapting the existing analysis. Consequently, the analyses can be used either in isolation (by omitting the transformations) or in the coupling (by executing the transformations).

Grey-box or *white-box* coupling are not suitable for our coupling goal because they require adapting the analyses to combine metamodels (white-box coupling) or functionality for coordinating internal steps (grey-box coupling). Consequently, these coupling forms require adapting the architectural analysis to cope with information originating from the implementation or by maintaining two versions of the architectural analysis. We discuss drawbacks of these coupling forms in relation to our black-box coupling selection in the following.

In *white-box* coupling the metamodels are combined. In our running example, this combination would require merging the PCM metamodel with details of the implementation like

information about encryption algorithms or encryption to detect vulnerabilities arising from the implementation. When omitting the architectural information about encryption (i.e., «*encrypted*» annotation) two versions of the analysis and its metamodels are necessary; the original version and one that includes the implementation details and functionality to detect how the implementation affects the architectural security. When including all architectural information and implementation details in the combined metamodel, the coupled analysis must be adapted to select whether the analysis should be performed based on the architectural design or the implementation details. For example, the analysis must be adapted to decide whether it should check whether data transmitted over a communication link is encrypted based on the «*encrypted*» annotation or based on the encryption algorithms represented in the combined metamodel. Furthermore, the coupled analysis must defer the detection of vulnerabilities arising from the implementation to the respective analysis which must, in turn, be able to interpret the combined metamodel. In both cases, the coupled analysis must be adapted to cope with information originating from the implementation. As stated before, this adaptation is not necessary in black-box coupling because it only operates on the input and output models of the analyses.

Merging the metamodels in white-box coupling also results in more complex metamodels which has negative impact on the maintainability of metamodels [123]. Furthermore, the information in the combined metamodel commonly originates from different roles in the software development process responsible for different development phases (e.g., software architects and component developers in component-based software development [275]). Consequently, the combined metamodel can only be applied by software engineers that are experienced with the domains of architecture, implementation and security (i.e., the roles of software architect and component developers have to be combined). In black-box coupling, the architectural models and implementation can still be created by the respective roles isolated in their dedicated development phase.

In *grey-box* coupling, the information exchanged (e.g., intermediate results) is commonly used in the computations of the coupled analyses. An exemplary scenario in our running example would be the calculation of the insecure deployment per component by the architectural analysis. Then, the architectural analysis could stop calculation until the information of source code analysis for the value of confidentiality levels resulting from the implementation for a specific component is available (e.g., the value *high* for the parameter *userData* in the service *store*). This approach would be feasible if one analysis requires a larger duration to compute its results than the other analysis. However, grey-box coupling requires analyses to include functionality for coordination of the internal steps and for the communication. This functionality is commonly provided by coupling frameworks (e.g., OPAL [125] for modular source code analyses or High-level Architecture (HLA) [57] for distributed simulations). Consequently, the architectural analysis can no longer be used in isolation without including additional functionality in the analysis tools. Examples for such functionality is the use of standard values when executed in isolation (e.g., values defined in the architectural specification). Such functionality again requires detecting whether the analysis is executed in isolation or in the coupling.

Black-box coupling also imposes another benefit regarding the modifiability of the analyses — a sub-characteristic of maintainability in the ISO/IEC 25010 standard [141] — compared to white-box and grey-box coupling. The applied analysis techniques and analysis tools can be freely modified or exchanged as long as the input and output metamodels are stable. For example, analyzers, can be modified to use other programming languages with the goal to increase analysis efficiency as illustrated by Boltz et al. [34]. In a grey-box coupling approach, reimplementing or adapting tooling would again require ensuring that the communication and coordination are realized correctly, e.g., the sequence when information has to be sent and received, to achieve the desired coupled behavior. Moving to other platforms or programming languages requires that a fitting coupling framework is available. It also requires updating analysis tooling when the coupling functionality or the API of the applied coupling framework changes. In white-box coupling, the adaptation of the analysis tooling must be performed while ensuring that the combined metamodel is still supported correctly.

Still, the black-box coupling also has drawbacks. The main drawback we see is the complexity in creating the transformations between the input and output models of the analyses in the coupling. For the transformations to be valid, a software engineer must be able to determine *valid* relations between elements in the input models and output models of the analyses. The validity of these relations depends on the syntax *and* semantics in the (meta)models of the analyses [323, 335]. Consequently, for a software engineer to determine these relations, the syntax and semantics of the analysis input and output models must be clear. For this to be clear, the software engineer must be experienced with the respective analysis models and tools or must interact with experts for these analyses. The relations between the input and output models are also not guaranteed to exist in every combination of analyses [70, 323]. This results in a software engineer investing effort for their detection which may not lead to a successful coupling.

Grey-box coupling or white-box coupling already requires involving the experts with the respective analysis models and tools. Thus, the availability of the respective expertise is guaranteed. Additionally, in grey-box or white-box coupling the analyses or their metamodels can be adapted to provide specific information. This contrasts the black-box coupling, which does not allow such modifications.

We discuss additional limitations of black-box coupling in Section 4.5.

4.1.3. Selecting the Composition Type

Black-box coupling requires selecting a *composition type* that allows achieving the goal of detecting architectural vulnerabilities arising from a non-compliant implementation based on the input and output models of the coupled analyses. In this dissertation, we apply the *sequential composition* type that parameterizes one analysis with the output of another analysis [323]. Due to the black-box coupling, we realize the sequential composition by modifying the input model of the architectural analysis with information retrieved from the vulnerabilities reported in the output model of the source code analysis.

As stated by Talcott et al. [324], the output of model-based analyses is an abstraction of the input model. Consequently, for achieving our coupling goal, directly relating the output of the source code analysis to the output of the architectural analysis requires bridging a larger semantic gap between the implementation and the architecture compared to relating the output of the source code analysis to the input of the architectural analysis. Bridging this gap between the output models would require replicating the logic of the architectural analysis to either remove existing architectural vulnerabilities or add new vulnerabilities. With sequential composition, our coupling defers the detection of the architectural vulnerabilities originating from the implementation to the architectural analysis rather than leaving it to the analysis coupling. Thus, the application of sequential composition reduces the complexity of bridging the gap between architectural security and implementation security.

We focus on specification-based non-compliance between the architectural design and the implementation. Consequently, in our coupling, we aim to parameterize the architectural analysis by integrating the values of a security characteristic resulting from the non-compliant implementation presented in the source code analysis output model. This proposed analysis coupling addresses uncertainty arising from *guesses* in the architectural security design (i.e., the values of the security characteristics described in the design phase) by incorporating information from the implementation [27]. This parameterization outlines the two main tasks of the coupling: Enabling the calculation of non-compliant values of security characteristics from the source code analysis output model and integrating these values into the architectural analysis input model. We discuss this task in more detail in Subsection 4.4.1.

A coupling realizing the composition types *Result Composition* or *Model Decomposition and Result Composition* [323] is not appropriate because of two reasons: First, both approaches require calculating the results of the analysis from a single model or parts of it. In the context of this dissertation, this is only possible if a model would comprise the architecture, implementation and the specifications for the respective analyses. We discuss drawbacks of such an approach in the context of white-box coupling in Subsection 4.1.2. Secondly, these approaches aim to combine the *output model* of the applied analyses. As we are interested in architectural vulnerabilities arising from non-compliance in the implementation, this modification would require adding vulnerabilities or removing existing vulnerabilities which is infeasible in our context as discussed before. Adding architectural vulnerabilities would require replicating the architectural analysis logic, which contradicts the idea of reusing existing analyses to determine the impact of a non-compliance on architectural security. Similarly, removing existing vulnerabilities would again require replicating the architectural analysis logic to determine whether a vulnerability is still valid given the information retrieved from the source code analysis.

4.2. Roles in Analysis Coupling

The sequential black-box coupling introduces the challenge of handling the complexity of connecting two independent analyses using different system abstractions based on their input and output models only. This task requires expertise in the independent analyses as well as knowledge about how to bridge the gap between them — especially involving the gap between the different system abstractions.

To address this complexity, we introduce a new role *coupling expert*. Furthermore, we extend the responsibilities of the roles *analysis architect* and *analysis user* proposed by Koch [169]. We adopt these roles from the context of decomposing and composing independent analyses from reusable analysis components [169] to the context of coupling independent analyses. The *analysis coupling expert* is responsible for connecting single analyses into a coupled analysis. We omit the role of *analysis component developer* [169] because we couple entire analyses rather than analysis components. A person can have multiple roles in the coupling process, e.g., the coupling expert can also be the analysis architect of one of the coupled analyses.

The *analysis architect* assembles analysis components into an independent analysis. This independent analysis can either be used in isolation or as part of a coupled analysis. In our analysis coupling, analysis architects develop the architectural analysis and source code analysis independently of each other. Usually, these are two different teams with no connection between them (e.g., no shared members exist, they are not in the same organization, or they do not share a common goal). Analysis architects create the analysis functionality and the languages for the input and output of the analysis. Consequently, the analysis architects are aware of the semantics of the input metamodels and the output metamodels.

The *analysis user* applies the independent analysis and the coupled analysis to analyze a system. In the context of this dissertation, the *analysis user* is a *software architect* that defines the input model for an architectural analysis to detect architectural vulnerabilities. For this purpose, the analysis user (1) creates the input model for the isolated architectural analysis, (2) applies the isolated architectural analysis to detect vulnerabilities, (3) defines the security characteristic for which non-compliance has to be investigated, and (4) applies the coupled analysis to detect new architectural vulnerabilities arising from non-compliance in the implementation.

The new role of *analysis coupling expert* (in the remainder of this thesis only called *coupling expert*) couples an architectural analysis and a source code analysis into a coupled analysis. The responsibility of the coupling expert is bridging the gaps between the input and output (meta)models of the analyses to be coupled. The coupling expert selects a source code analysis that can analyze non-compliance regarding the security characteristic in the architectural analysis used by the analysis user. We support the selection process with our contribution **C3** (presented later in Chapter 6), which allows reasoning about the capability of source code analyses to analyze non-compliance regarding security-related properties specified in a security DSLs. However, there are other factors influencing

the selection of a source code analysis, such as the quality properties they expose. In a case where several analyses are capable of analyzing non-compliance regarding the security characteristic in the architectural analysis, these properties can be used to support the selection process for the needs of the analysis user. For example, there are fully automated analyses (e.g., JOANA [98] or FlowDroid [19]) or semi-automated analyses (e.g., the verification approach of Greiner et al. [105]) that can detect non-compliant information flows in the implementation. These analyses have varying accuracy and different degrees of automation. Therefore, the coupling expert interacts with the analysis user to select a source code analysis providing the required properties. The coupling expert then creates the transformations between the analyses as part of the black-box coupling. The coupling expert communicates with the analysis architects of both analyses to be coupled to determine the relations between the input and output models of the analyses and to verify the correct realization of the transformations. In contrast to the analysis architect, the coupling expert has a black-box view on the analyses and only needs to focus on input and output (meta)models and relations between these (meta)models.

4.3. Semantics in Analysis Coupling

Analysis coupling requires special attention to the semantics of the information exchanged between the analyses to avoid calculating wrong results due to misinterpretation [323, 335]. This is especially important in a black-box coupling where the coupling is established based on transformations between the input and output models of the analyses. These transformations use relations between the elements in the input and output models of the analyses, determined by the coupling expert. Consequently, a coupling expert must ensure that the relations between the *syntactic elements* in the input and output models of the analyses are semantically correct [323, 335]. We support the task of the coupling expert by proposing three types of semantic relations between syntactic elements in Subsection 4.3.1.

To make use of semantic relations in the coupling, they have to be represented on the syntactic level. To support this task, we state the hypothesis that *we can apply (meta)models to determine semantic relations between syntactic elements*. With this hypothesis, metamodels can be used as syntactic representations of the semantic relations to be exploited for automated coupling. We discuss this hypothesis in Subsection 4.3.2.

The transformations used in the coupling are defined on the metamodel level [238, 317]. However, only considering the metamodel level for defining transformations can be challenging for the coupling expert as we found that metaclasses in security analyses can expose different semantics when instantiated. For example, the metaclass *Security Characteristic* in the metamodel of the Data Flow Analysis (DFA) of Seifermann et al. [300] can be instantiated to represent different security concepts (e.g., *confidentiality levels of data*, *roles that are allowed to access data* or even *locations of resources*). Consequently, only formulating transformations on the metamodel level without considering the semantics of the instances can lead to incorrect transformations in the coupling. We discuss the

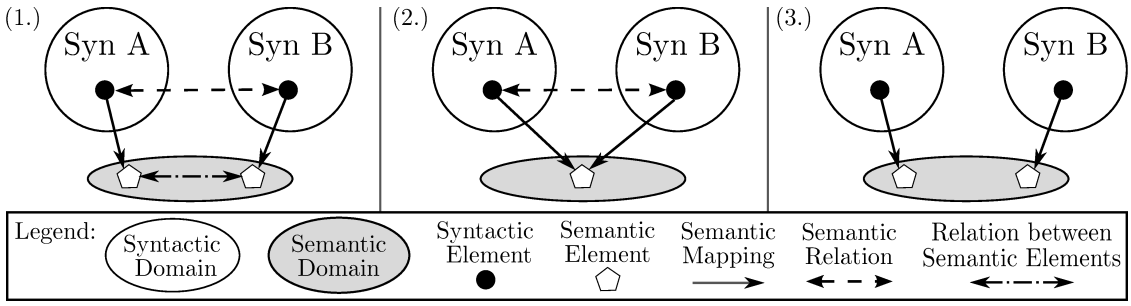


Figure 4.1.: Three types of semantic relations relevant for analysis coupling. (1.) shows the semantically related relation, (2.) shows the semantically identical relation and (3.) shows the semantically unrelated relation.

capability of a metaclass exposing different semantics on the instance level further in Subsection 4.3.3.

4.3.1. Semantic Relations in Analysis Coupling

We define a *semantic relation* between two *syntactic elements* of different (meta)models based on the relation between the *semantic elements* they are mapped to by the semantic mapping (see Subsection 2.1.3). In this context, we define three relevant types of semantic relations: *semantically related*, *semantically identical* and *semantically unrelated* as illustrated in Figure 4.1.

We use a set-theoretic view of the *syntactic* domain and *semantic* domain to describe the semantic relations. Viewing those domains as sets of elements is analog to our view on metamodels and models as sets of metaclasses and instances of these metaclasses (see Section 2.1). We do not discuss a more detailed view of the semantic domain in this thesis because it suffices to express our concepts, and because it is not the main research focus of this dissertation. We simplify by assuming that the syntactic elements map to semantic elements in one semantic domain, but the concepts can also be transferred to multiple semantic domains. This simplification is possible because semantic domains are composable [115].

The semantically related relation type and the semantically identical relation type form a *semantic overlap* between the information of the analyses. This overlap is necessary as otherwise, there is no information common in the analyses which can be used for the coupling. Our notion of semantics and semantic overlap is similar to the notion of *properties* and their overlap described by David et al. [61]. The properties mentioned by David et al. [61] are used to classify syntactic elements in the system. For example, David et al. [61] describes the property of a line follower robot to be *safe* or *unsafe*. This information is provided in the context of security analyses by annotating security characteristics to system elements (e.g., data is annotated with the property *confidential* or *non-confidential*). Still, we use the terminology of semantics and semantic overlaps to highlight the relation between a syntactic element and its semantic meaning — including, amongst others,

security characteristics and their annotation — rather than the actual property which is annotated to the system element. For example, *Confidentiality Level* in the architectural model and *Security Level* in the source code model of the running example both express the same semantic meaning of a confidentiality classification for data. Similarly, these security characteristics can be annotated with differently named syntactic elements (e.g., *Level* in the architectural model and *InformationFlow* in the source code model) which again express the same semantic meaning of an annotation that provides confidentiality classification to system elements. With the terminology of semantics and semantic overlaps, we highlight the reasoning of this relation (e.g., two syntactic elements describe the same concept) instead of highlighting the actual property. With this terminology, we aim to support the coupling expert in determining the relations between syntactic elements of the analyses to realize the coupling. However, we acknowledge that the terminology of properties and their overlap as used by David et al. [61] can also be used for classifying syntactic elements and their annotations in the context of security analyses interchangeably.

Semantically Related In the *semantically related* (*SR*) relation type, syntactic elements syn_A and syn_B in their respective (meta)models are mapped to different semantic elements, i.e., $SM(syn_A)$ and $SM(syn_B)$ ($SM\colon Syn \rightarrow Sem$ describes the semantic mapping; see Subsection 2.1.3). Between these semantic elements a *relation* in the semantic domain $\langle SM(syn_A), SM(syn_B)\rangle_{Sem}$ exists as captured in Equation 4.1.

$$\begin{aligned} &\forall syn_A \in Syn_A, syn_B \in Syn_B\colon \\ SR(syn_A, syn_B) \iff\ & SM(syn_A) \neq SM(syn_B) \wedge \langle SM(syn_A), SM(syn_B)\rangle_{Sem} \end{aligned} \tag{4.1}$$

For example, the syntactic elements *Threat* and *SecurityObjective* belong to two different metamodels used by analyses to be coupled. Both syntactic elements are mapped to semantic elements describing the concept of a *threat* and a *security objective*, respectively. In the semantic domain, we can capture a relation *threatens* to describe that a *threat* threatens a *security objective*. Consequently, the syntactic elements *Threat* and *SecurityObjective* are semantically related by the relation *threatens* in the semantic domain.

Semantic relations can also be transitive. The relation $\langle SM(syn_A), SM(syn_B)\rangle_{Sem}$ can be established by a transitive relation over the semantic element $sem_j \in Sem$ if a semantic relation between $SM(syn_A)$ and sem_j, as well as between sem_j and $SM(syn_B)$ exists: $\langle SM(syn_A), sem_j\rangle_{Sem} \wedge \langle sem_j, SM(syn_B)\rangle_{Sem} \Rightarrow \langle SM(syn_A), SM(syn_B)\rangle_{Sem}$.

Semantically Identical The *semantically identical* (*SI*) relation type is a specialization of the semantically related type, where the relation on the semantic domain describes the identity as shown in Equation 4.2. Thus, in the semantically identical type, syntactic elements are both mapped to the same semantic element (i.e., $SM(syn_A) = SM(syn_B)$).

$$\forall syn_A \in Syn_A, syn_B \in Syn_B\colon SI(syn_A, syn_B) \iff SM(syn_A) = SM(syn_B) \tag{4.2}$$

This relation occurs in architectural models and source code when elements with the same semantics are available, e.g., *OperationSignatures* in the PCM [275] and *Methods* in Java

as described by Langhammer et al. [181]. In the security domain, an example for this semantic relation is provided by Katkalov et al. [159], where *security domains* annotated to UML [235] diagrams are mapped to instances of *Security Level* of *JOANA* [98], both representing the description of a security classification for data.

Semantically Unrelated In the semantically unrelated relation type, two syntactic elements syn_A and syn_B are mapped to different semantic elements, without any relation between these semantic elements. We address this relation type because deciding whether two syntactic elements are semantically related or unrelated may depend on experience of the coupling expert. A coupling expert can determine the syntactic elements to be *semantically unrelated* (e.g., due to a lack of knowledge), while another coupling expert may be able to find a valid relation between the mapped semantic elements. Consequently, there may also be a shift between the semantically unrelated type and either semantically related or semantically identical types over time as the coupling expert gains additional insights.

4.3.2. Calculating Semantic Relations with Metamodels

Automating the coupling requires defining how the respective semantic relation types can be represented in a format that can be processed by a coupling approach. We propose using metamodels that abstract semantic relations between elements in the (meta)models of the analyses to be coupled. Based on our definition of the semantically related type before, such a metamodel has to fulfill the following requirements: First, the metamodel contains metaclasses with defined semantics to which semantically correct mappings for the model element a and the model element b can be defined. Secondly, the metamodel contains references that allow representing a path between the metaclasses representing the semantic elements. This metamodel can then be instantiated to represent the syntactic elements a and b and the semantic relation between them. For operationalization, the instance of this metamodel has to be traversed to detect the path between the syntactic elements of the models used in the coupling.

In the case of the semantically identical type, a metamodel suffices that contains either a metaclass to which syntactic elements of the input and output (meta)models can be mapped, or two metaclasses connected by a reference which describes the identity relation. The path calculated between the elements of the models used in the coupling is then to search for the element mapping to the same instance of the metaclass or to traverse the reference describing the identity relation. Examples of such metamodels are *correspondence (meta)models* (e.g., used by [120, 147]) or triple-graph grammars [297] that define correspondences between elements of different (meta)models.

We discuss examples of the (meta)models capturing the semantic relations for the semantically related and semantically identical types in our coupling approaches presented in Chapter 5.

4.3.3. Challenge in Analysis Coupling: Model-Level Semantics

In our research, we found analyses using syntactic elements which allow exposing different semantics when instantiated. Examples are metaclasses that can be used to describe different security classifications of data such as *Confidentiality Levels* or *Roles* (e.g., used in [98, 159, 224, 300]). We call the capability of a metaclass to be refined by the semantics of its instances *model-level semantic refinement*.

In this refinement, a *refinement relation* between two semantic elements exists which expresses that the refining semantic element has more specific semantics than the refined element. For example, the metaclass *Security Level* of *JOANA* [98] can be mapped to the semantics of an element representing security classifications of data. The semantic element *Role* and *Confidentiality Level* refine this semantic by expressing that the classifications represent security based on roles of actors or confidentiality levels (e.g., with the values *high*, *low*) respectively. This relation is similar to the incarnation mapping of Konersmann et al. [171], where an element of the *specific model* incarnates an element of the *reference model*. *Model-level semantic refinement* introduces challenges for the coupling expert regarding creating the transformations in the coupling, and selecting appropriate analyses.

We illustrate these challenges by adapting the analyses of our running example. Here, we assume that *JOANA* [98] can only analyze information flows using the values *high* and *low*, or disjunctive roles. This capability is similar to the Access Analysis [172]. Such a restriction is realized, for example, by integrating the *security policy* (i.e., the definition when a security vulnerability occurs) into the analysis. In contrast, the *Security Characteristic* metaclass in the metamodel of the DFA presented by Seifermann et al. [300] allows the description of different security-related characteristics for architectural elements. Seifermann et al. [300] instantiates this metaclass to express the semantics of *confidentiality levels*, *roles* or *locations*.

In literature, access restrictions can be defined by composing basic roles, such as in the example provided by Tuma et al. [338] or the *TravelPlanner* system [159, 211]. The example provided by Tuma et al. [338] provides the basic values *Alice* and *Bob* and the combined value *Alice;Bob*. In this example, the security policy defines that flows are allowed from data classified as either *Alice* or *Bob* to program locations classified as *Alice;Bob*. Thus, this policy indicates a *conjunctive* composition of the roles because *Alice;Bob* implies higher confidentiality by requiring the permission to access information from *Alice and Bob*. In contrast, the *TravelPlanner* system [159, 211] provides, amongst others, the basic values *Airline*, *User* and *TravelAgency*. In this system, a flow is allowed, for example, from data classified as *Airline;User* to locations classified as *User*. This policy indicates a *disjunctive* composition because *Airline;User* implies lower confidentiality by requiring access to either *Airline* or *User*. This example shows that the same syntactic delimiter (;) can indicate semantically different compositions of the values (i.e., *disjunctive* or *conjunctive*) depending on the security policy used.

In this example, the coupling expert must have the knowledge about details of both analyses (i.e., the expressible security policies) to understand the possible semantics of

the delimiter. This requirement is especially important if the security policy is integrated into the analysis. Given the example with the combination of roles, the coupling expert may interpret a delimiter as indicating a disjunctive composition, while the analysis actually interprets it as a conjunctive composition. Such a misunderstanding can lead to a semantically incorrect transformation in the coupling. For example, the coupling expert may define a transformation that maps the value *Alice;Bob* in the architectural analysis to *Alice;Bob* in the source code analysis, assuming that both analyses interpret the delimiter as conjunctive. Consequently, the source code analysis may calculate that a flow from data classified as *Alice;Bob* to locations classified as *Alice* is allowed and does not report any violations while the architectural analysis would interpret such constellations as not allowed.

The example also illustrates that the refinement relation constrains the expressiveness of the architectural analysis used in the coupling *on the model level*. The input model of the architectural analysis must only be used to express semantics that can also be expressed in the source code analysis. In our example, the architectural analysis must only be applied to represent *Confidentiality Levels* and *disjunctive Roles* but not *conjunctive* roles due to the capabilities of the source code analysis.

In this thesis, we only aim to highlight the challenges of model-level semantic refinement for the coupling expert. Providing a systematic approach to support the coupling expert in dealing with these challenges is out of scope of this thesis and is part of our future work (see Section 11.3).

4.4. Overview of the Coupling Process

The coupling process describes the steps necessary to realize the sequential black-box coupling with the goal of detecting architectural vulnerabilities arising from an implementation which is non-compliant with the architectural specification. For realizing this coupling, the coupling process (illustrated in Figure 4.2) uses one architectural analysis and one source code analysis to parameterize the architectural analysis with the values of one security characteristic under investigation which result from the implementation. We call the triple of architectural analysis, source code analysis and security characteristic under investigation the *coupling scenario* (see Definition 4).

Our sequential black-box coupling parameterizes the architectural analysis based on the results of the source code analysis. In Subsection 4.4.1, we discuss the main tasks for the parameterization that have to be performed by the coupling process. The input and output models of the analyses are the main artifacts to achieve the black-box coupling (see Subsection 2.5.1). Thus, before we can detail the coupling process, we introduce the (meta)models involved in Subsection 4.4.2. Finally, Subsection 4.4.3 describes the process steps of the coupling process.

The coupling design presented in this chapter outlines the models used in the coupling and coupling process steps. However, the actual information in the (meta)models used and

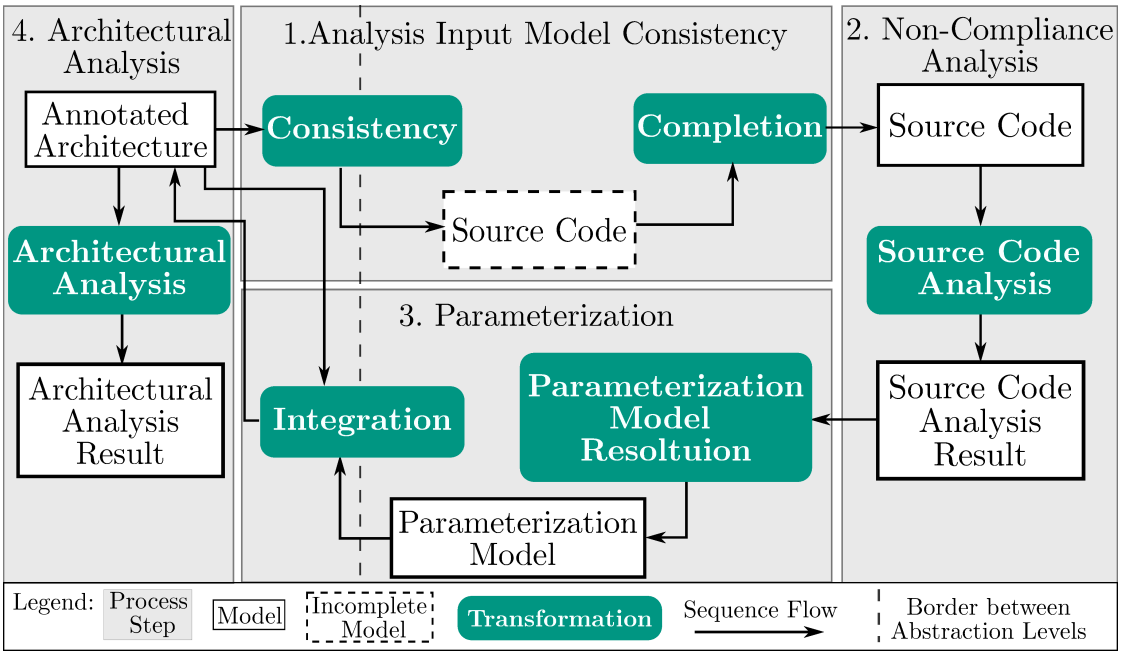


Figure 4.2.: Overview of the process steps in the proposed coupling approach with the comprised models and transformations. Roles are excluded for better readability.

the realization of the process steps depends on the types of coupled source code analyses and the security characteristic under investigation. This information also influences the semantic relations existing between the (meta)models in the coupling process. Consequently, the (meta)models and process steps have to be specialized for the specific coupling approaches which instantiate the coupling design. We describe examples for such specific coupling approaches in Chapter 5.

4.4.1. Parameterization

The goal of this dissertation is to detect architectural vulnerabilities that arise from an implementation which is non-compliant with the architectural security design. For this purpose, we parameterize the architectural analysis input model by representing values of a security characteristic resulting from the implementation (see Section 4.1). With this approach, the architectural analysis can detect vulnerabilities based on the realized system (i.e., the values resulting from the implementation), rather than based on values assumed in the early design phase.

For this parameterization, our coupling process has to perform two tasks: First, the process has to evaluate non-compliance in the implementation regarding the security characteristic under investigation in the architectural specification. We call this task *non-compliance detection*. According to our definition of non-compliance (Definition 1), this task requires calculating the value of the security characteristic resulting from the implementation for an implementation element. Because the source code analysis only reports vulnerabilities, these values have to be *derived* from the source code analysis

output model. Once these values are calculated, they have to be compared with the values specified in the architectural analysis input model. In general, this comparison requires determining conformance between the values resulting from the implementation and the value specified in the architectural analysis input model. However, this task can be omitted if all reported vulnerabilities in the source code analysis indicate non-compliance. As we will show in our coupling approaches (Chapter 5), this is the case when source code analyses report vulnerabilities based on values of security characteristics that correspond with the security characteristic of the coupling scenarios.

Secondly, when non-compliance is detected, the specification in the architectural analysis input model has to be modified to reflect the value resulting from the implementation rather than those specified in the architectural specification. This task is called *parameterization*. The modification depends on how the security characteristic of the coupling scenario is represented in the specification. In our running example (Chapter 3), different value ranges are described for the security characteristic *Confidentiality Level* and the security characteristic *Encryption* in the architectural analysis. *Confidentiality Level* comprises the values *low*, *medium*, *high* to classify the confidentiality level of data. For the security characteristic *Confidentiality Level*, the value corresponding to the value resulting from the implementation has to be set. In contrast, *Encryption* is described by whether data is *encrypted* or *not encrypted* (binary value). These values are represented by the existence or absence of the specification «*encrypted*» for communication links. Representing the value resulting from the implementation for *Encryption* can be realized by removing or maintaining the annotation in the architectural analysis input model.

These tasks result in the need for the coupling process to calculate two types of information: (1) the values resulting from the implementation and (2) the specification of the values of the security characteristic under investigation for dedicated architectural elements.

Calculating the Values Resulting From The Implementation The values of the security characteristic resulting from the implementation have to be calculated for implementation elements from the source code analysis output model. However, source code analyses are designed to report vulnerabilities in the implementation rather than to provide the values of a security characteristic resulting from the implementation. Thus, the source code analysis output model has to be interpreted to extract the values of the security characteristic resulting from the implementation for an implementation element.

We illustrate this interpretation on our running example (Chapter 3) for the two information flows to the parameter *userData* in the method *store*. These two flows originate from the field *privilegeSum* in the *OnlineShop* class and from the parameter *userData* in the method *buy*. The interpretation for these flows can be performed as follows: Due to the flows, data classified with the value *high* or *medium* are allocated in *userData* of the method *store*. Based on the security policy of our running example, an attacker that is able to compromise the confidentiality of data classified with the value *medium* would also violate the confidentiality of data classified with the value *high* but not vice versa. Thus, we interpret in our running example that the implementation results in the value *high* for the

parameter *userData* in the method *store*. Other security policies and security characteristics require different interpretation depending on their semantics. Therefore, we cannot provide a comprehensive generalization for all possible security characteristics and source code analyses. However, we discuss the interpretations for specific security characteristics in our coupling approaches (Chapter 5) and formulate future work in Section 11.3 for generalizing these interpretations.

Calculating the Specifications of Security Characteristic of the Coupling Scenario Our coupling process requires calculating the specification of the values of the security characteristic for architectural elements that are (1) potentially affected by non-compliance (in the *non-compliance detection* task) or (2) that are determined to be non-compliant (in the *parameterization* task). We assume that the values of the security characteristic in the architectural analysis input model are specified using annotations (see Subsubsection 2.3.2.2).

For calculating these annotations, the architectural element corresponding to the implementation element potentially affected by non-compliance must be determined. However, this architectural element alone does not suffice to perform the parameterization. In our analysis coupling, we use additional information (amongst others the security characteristic of the coupling scenario) and assumptions to calculate the annotation. This information must be defined by a coupling approach so that identification of all annotations affected by non-compliance is possible.

For example, an architectural model may use various annotations annotating different security characteristics for the same architectural element (e.g., the annotations *Location*, *Sharing* and *TamperProtection* for the architectural element *Resource Container* in the Access Analysis [172]). Consequently, we have to identify the annotation using the security characteristic of the coupling scenario to perform the parameterization. However, analyses can also use several annotations, annotating the same security characteristic. The source code analysis *JOANA* [98], for example, uses the annotations *Source* and *Sink*. Both annotations annotate the security characteristic *Security Level* to fields or parameters (omitted in our running example for simplicity). For this case, we have to make the assumption that all annotations annotating the same security characteristic to an architectural element annotate the same values.

4.4.2. Models in the Coupling Process

Our coupling process uses the black-box coupling form to realize the coupling. This coupling form implies that the coupling process focuses on models used by the analyses and transformations between them (see Subsection 4.1.2). We will discuss the models applied in the coupling process in the following (summarized in Table 4.1).

The main artifacts used are the input model of the architectural analysis (*Annotated Architecture*), the output model of the architectural analysis (*Architectural Analysis Result*),

Model Name	Symbol	Description
Annotated Architecture	A	Model used by the architectural analysis to calculate vulnerabilities. Comprises the architectural design and the architectural specification
Architectural Analysis Result	R_A	Model representing the vulnerabilities detected by the architectural analysis
Source Code	C	Model used by the source code analysis to calculate vulnerabilities. Comprises the system implementation. Also includes specifications in case of specification-based analysis
Source Code Analysis Result	R_C	Model representing the vulnerabilities detected by the source code analysis
Parameterization Model	PM	Model comprising all information necessary for parameterization

Table 4.1.: Summary of model names in the coupling process, their formula symbols used in this dissertation and the corresponding descriptions.

the input model of the static source code analysis (*Source Code*) and the output model of the source code analysis (*Source Code Analysis Result*). We discuss these models in more detail in Subsubsection 4.4.2.1.

Additionally, we introduce the intermediate model *Parameterization Model* to address the challenge of interpreting the *Source Code Analysis Result* in order to extract the necessary information for parameterization. This challenge arises because the *Source Code Analysis Result* represents vulnerabilities in the source code rather than the values of security characteristics resulting from the implementation. Thus, the coupling expert has to extract information necessary for the parameterization from the *Source Code Analysis Result* (see Subsection 4.4.1). Processing of the *Source Code Analysis Result* and extraction of the necessary information requires knowledge mainly available to the analysis architect of the source code analysis. In contrast, the coupling expert is responsible for connecting the architectural level and the implementation level. We introduce the *Parameterization Model* in Subsubsection 4.4.2.2.

The actual content of the (meta)models in the coupling process depends on the types of analyses to be coupled and the security characteristic under investigation. Thus, addressing this content of the (meta)models is part of the specific coupling approaches presented in Chapter 5. To avoid coupling approaches that are tailored to specific analyses, their (meta)models and their security characteristics, we describe elements necessary to perform the coupling with reference metamodels. We introduce the concept of reference metamodels in more detail in Subsubsection 4.4.2.3. These reference metamodels present an interface for the coupling of *types* of analyses and *types* of security characteristics rather than specific analyses and specific security characteristics.

The (meta)models to capture the semantic relations depend on the types of semantic relations which are specific to the information in the input and output models of the analyses in the coupling and the security characteristic under investigation. Consequently, we introduce the *Annotated Architecture*, the *Architectural Analysis Result*, the *Source Code*, the *Source Code Analysis Result* and the *Parameterization Model* in the following and defer the description of the (meta)models to capture the semantic relations to the specific coupling approaches in Chapter 5.

4.4.2.1. Input and Output Models of Analyses in the Coupling

Here, we introduce the input and output models of the analyses used in the coupling process. We discuss the *Annotated Architecture* and *Source Code* jointly as both represent the system to be analyzed. The metamodels of these models are developed by the analysis architects of the respective analyses.

Annotated Architecture* and *Source Code The *Annotated Architecture* (A) and *Source Code* (C) represent different system abstractions and comprise the information necessary to perform the analyses. The *Annotated Architecture* contains the architectural design and the specification of security characteristics for the architectural elements. For example, the PCM [275] or UML [235] component-diagrams include various structural information such as components, provided and required services, connections between these services, resources used to deploy components or connections connecting the resources. We focus on architectural analyses which specify at least one security characteristic for the structural elements of the software architecture (see Subsection 4.1.1). Architectural analyses exist which comprise behavior descriptions for calculating vulnerabilities (e.g., the DFA of Seifermann et al. [300] or UMLSec [154]). These behavior descriptions are commonly on a higher level of abstraction than the implementation. The SEFFs in the PCM used by DFA, for instance, can use an *Internal Action* to express the encryption of data without specifying the encryption algorithm. Our coupling approach supports such analyses if they also annotate structural elements. However, we do not consider the behavior descriptions of such analyses for the coupling. In contrast, the *Source Code* focuses on describing the detailed behavior of the system in the form of method implementations or fields. Although the focus is on behavior description, the *Source Code* also includes information to structure it such as packages, classes, and methods.

The content of the *Source Code* differs based on the types of applied static analyses. In this dissertation we focus on two major types of static source code analyses: pattern-search-based analyses and specification-based analyses (see Section 2.4). Pattern-search-based source code analyses only use the source code as input which is investigated for patterns that indicate a vulnerability. Specifications can exist in the source code for describing various information (e.g., override annotations in Java or Spring Framework annotations [109]). However, these specifications are not required to perform the pattern-search-based analysis. Consequently, we do not consider these specifications as part of the *Source Code* in the coupling with pattern-search-based analyses.

The *Source Code* for specification-based source code analyses comprise the source code of the systems and specifications for implementation elements to investigate whether the implementation conforms to the specified properties. Similarly to architectural analyses, the specifications define the properties of the implementation elements expected to be realized in the final system. Our running example exemplifies these specifications by the annotation of the value *low* to the parameter *userData* in the method *store*. This annotation specifies that the implementation should be created so that this parameter only contains non-confidential data. When discussing our coupling approach using specification-based source code analysis in Section 5.1, we call the source code analysis input model *Annotated Source Code* instead of *Source Code* to highlight the focus on the specifications.

Source Code Analysis Result The *Source Code Analysis Result* (R_C) represents vulnerabilities detected by the source code analysis in the analyzed *Source Code*. Applying a black-box coupling implies that the information in the *Source Code Analysis Result* requires a semantic relation to the information in the *Annotated Architecture* to achieve the tasks presented in Subsection 4.4.1. Thus, for our coupling, we expect the *Source Code Analysis Result* to contain elements that can be related to elements of the *Annotated Architecture* rather than quantitative values or boolean values stating whether the system is secure.

However, the information in the vulnerabilities and the representation of this information varies based on the analyzed property and specific analysis. Even for analyses of the same security property such as information flow security, variances in the information exist. The information flow analyses FlowDroid [19] and CodeQL [100, 214], for instance, report whether an information flow exists between a source and a sink without using security characteristics. In contrast, the information flow analyses *JOANA* [98] and JFlow [224] use security characteristics. This difference in information can arise even when standardization endeavors exist, such as the RS^3 Information-Flow Specification Language (RIFL) language [82] for information flow analyses. Such variances of information also exist in pattern-search-based source code analyses. Pattern-search-based analyses commonly report the locations and the patterns identified in their *Source Code Analysis Result*. However, *Find Security Bugs* [18] illustrates a case where an analysis omits the weakness associated to the pattern in the *Source Code Analysis Result*.

The *Source Code Analysis Result* also varies in the syntax used for representing information. For representing the security-relevant information, the analysis tools *Snyk* [310] and *Find Security Bugs* [18] both identify patterns that indicate usage of weak cryptographic algorithms. However, *Snyk* reports the pattern by the identifier *java/InsecureCipher* when the insecure encryption algorithm *DES* is used, and *Find Security Bugs* reports it by the identifier *DES_USAGE*. This variance can also be observed in how implementation elements are identified. The information flow analyses CodeQL [100, 214], *JOANA* [98] and FlowDroid [19], for instance, identify parameters as implementation elements in their *Source Code Analysis Result* with a different syntax. CodeQL identifies parameters by the location of the Java file the source code is located, the line and column of the parameter, and the parameter name and type. JOANA reports the class, method name, the parameter name and its type in byte-code like syntax. FlowDroid [19] provides the same

information but in a java-like syntax. The same variances exist in how pattern-search-based analyses represent the locations of the detected patterns. *Snyk* [310] reports the class and the line of code where the pattern is found. In contrast, *Find Security Bugs* [18] reports the class, the method name and lines of code relative to the method. There are endeavors to standardize the representation of source code analysis results such as the *Static Analysis Results Interchange Format (SARIF)* [16]. However, use of these standards is not mandatory.

Architectural Analysis Result The *Architectural Analysis Result* (R_A) represents architectural vulnerabilities arising when analyzing the *Annotated Architecture*. With the *Architectural Analysis Result* in our coupling process two insights can be obtained: First, the *Architectural Analysis Result* shows the architectural vulnerabilities that arise due to non-compliance. Secondly, the *Architectural Analysis Result* also provides insights into whether existing non-compliance impacts the architectural security. The running example in Chapter 3 illustrates the second case with the information flow between the parameter *userData* in the *buy* method and the parameter *entry* in the *log* method. This information flow results in non-compliance because now *high* classified data is allocated in the parameter *entry* classified to contain *low* data. However, this non-compliance does not lead to an architectural vulnerability (illustrated by the absence of a vulnerability involving the *Logger* in partial model (i) of Figure 3.3). No vulnerability is reported because the *Logger* component is deployed on a resource annotated with the *location EU* which does not violate the architecture security policy (partial model (b) of Figure 3.1).

In this dissertation, we simplify by assuming that every vulnerability reported by the architectural analysis is fixed before evolving the implementation — i.e., the *Architectural Analysis Result* does not contain vulnerabilities before applying the coupling. With this simplification, we focus on the detection of architectural vulnerabilities arising from non-compliance rather than on how to trace back the architectural vulnerabilities to the implementation. However, we are aware that achieving complete security is commonly not possible due to resource constraints [126, 203]. This is why we state future work in Section 11.3 to address the tracing between architectural vulnerabilities and vulnerabilities reported from the source code analysis. This tracing would allow determining whether architectural vulnerabilities arise from the architectural design itself or from non-compliance.

Nevertheless, the *Architectural Analysis Result* is an essential part of the coupling because only through the reported architectural vulnerabilities, software engineers can determine the impact of non-compliance on the architectural security. However, the *Architectural Analysis Result* is not involved in *establishing* the coupling. Thus, the discussion of the *Architectural Analysis Result* in the remainder of this thesis is limited.

4.4.2.2. *Parameterization Model*

Due to the variances in the information and syntax in the *Source Code Analysis Result*, the coupling expert faces challenges to perform the parameterization tasks presented in Subsection 4.4.1. This is why we introduce the *Parameterization Model* (*PM*) to address these challenges for the coupling expert. The *Parameterization Model* provides a uniform abstraction of the information in the *Source Code Analysis Result* for specific types of analyses that are required to perform the parameterization. For this purpose, the *Parameterization Model* contains only those elements that are necessary to perform the tasks presented in Subsection 4.4.1. Consequently, a source code analysis can only be applied in the coupling if a transformation from its *Source Code Analysis Result* to the *Parameterization Model* can be created. The *Parameterization Model* is created by the coupling expert as they know about the information necessary to perform these tasks.

Consequently, the *Parameterization Model* defines an interface for the analysis coupling. This interface enables (1) the analysis architects to focus on the interpretation and parsing of the *Source Code Analysis Result* as they know precisely which information they have to provide for the coupling and (2) the coupling expert to focus on creating the transformation to bridge the gap between the architectural analysis and source code analysis rather than parsing and interpreting the *Source Code Analysis Result*.

The necessity of the *Parameterization Model* arises from two challenges the coupling expert faces when interpreting the *Source Code Analysis Result*. First, the source code analyses are created to report examples for vulnerabilities in the implementation rather than to report values resulting from the implementation. Consequently, the coupling expert has to evaluate whether a *Source Code Analysis Result* contains enough information to determine non-compliance for a coupling scenario. For this evaluation, the coupling expert must investigate whether vulnerabilities reported contain enough information (1) to derive the values of the security characteristic resulting from the implementation for implementation elements, and (2) to calculate the respective annotations in the *Annotated Architecture*. For the patterns of pattern-search-based source code analyses, for instance, the *coupling expert* has to understand which pattern implies a specific weakness. This knowledge is necessary to relate the pattern to the security characteristic in the coupling scenario. In our running example, the coupling expert has to understand that the pattern representing the usage of *DES* (*DES_Usage*) violates the security characteristic *Encryption*. Understanding this relation is not always guaranteed as shown by challenges of software engineers in applying encryption [117, 226]

This example also shows the second challenge: the coupling expert must be capable to parse and understand the varying syntax in the *Source Code Analysis Result*. Using lines of code to identify implementation elements in the *Source Code Analysis Result* is reasonable as it allows to navigate the source code precisely. However, this representation requires the coupling expert the capability to parse these lines of code and to relate them to implementation elements. Errors in this parsing can lead to flagging wrong elements as affected by a vulnerability. Similarly, the coupling expert has to understand that identifiers

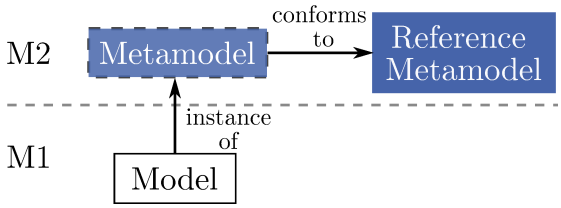


Figure 4.3.: Relations between reference metamodels, metamodels of analyses and their models. Includes metamodel layers from Figure 2.1 to illustrate that the reference metamodels are defined on the same metamodel level as the metamodels of the analyses.

java/InsecureCipher and *DES_USAGE* are reported for the same source code pattern: the usage of *DES* in a cryptographic Application Programming Interface (API).

The analysis architect has the knowledge about the information in the *Source Code Analysis Result*, the syntax used for its representation and which other information can be derived from it. Based on this premise, the *Parameterization Model* addresses these challenges by allowing the coupling expert to define the information necessary for the parameterization which is provided by a transformation created by the analysis architect.

4.4.2.3. Reference Metamodels

Our goal in this dissertation is to provide a coupling design and coupling approaches which generalize to analyses of the same *type*. Providing a coupling approach for all types of analyses and security characteristics is not feasible due to their large variety. Thus, the coupling approaches must support specific *types* of analyses and specific *types* of security characteristics rather than to be described for specific analyses or specific security characteristics.

We abstract from specific analyses and security characteristics by describing reference metamodels. These reference metamodels comprise the information necessary to perform the coupling for specific *types* of analyses and security characteristics. These reference metamodels are described for the metamodels involved in our coupling process (i.e., *Annotated Architecture*, *Source Code*, *Source Code Analysis Result* and *Parameterization Model*). The information necessary for the coupling is captured in form of metaclasses and references between the metaclasses relevant for a coupling approach. However, a coupling is only valid if the relations between the information required and provided by the analyses are correct on the syntactic *and* semantic level [323, 335]. Thus, we also expect that a description of a reference metamodel includes the expected semantics of its elements.

Consequently, an analysis can be used in the coupling only if it conforms to the reference metamodels; otherwise it does not comprise the information necessary for the coupling. To support the coupling expert in determining whether an analysis can be used in the coupling, we define when a metamodel of an analysis conforms to the respective reference metamodel in the following. For this description, we comprehend the reference metamodels as underspecified, i.e., the specific metamodels of the analyses can comprise metaclasses

and references not represented in the reference metamodels. This approach is similar to Konersmann et al. [171], which define that specific models can conform to reference models, although they contain additional model elements. Figure 4.3 illustrates the relations between reference metamodels, metamodels of analyses and their models. Specifically, the figure shows that we comprehend the reference metamodels to be allocated on the same metamodel level as the metamodels of the analyses.

We define a metamodel to conform to the respective reference metamodel, if for every metaclass in the reference metamodel, a conforming metaclass in the respective metamodel of an analysis exists. We define the conformance between a metaclass of the reference metamodel and a metaclass of a metamodel of an analysis in Definition 6.

Definition 6 ***(Conformance of a Metaclass of Metamodels to a Metaclass of the Reference Metamodel)***

Let M be an input or output metamodel of an analysis and $m, m' \in M$ be its metaclasses. Also let R be an input or output reference metamodel and $r, r' \in R$ be its metaclasses. Then a metaclass m in the input or output metamodel of an analysis conforms to a metaclass r in the reference metamodel if a valid transformation can be defined. As formally described in Equation 4.3, we define the transformation to be valid if

1. *The metaclass m fits the semantics of the metaclass of the reference metamodel r.*
2. *For every reference of the metaclass in the reference metamodel r to another metaclass r', the metaclass m must provide a (possibly transitive) reference to another metaclass m' conforming to r'. We define a reference as transitive if metaclass m' is reachable through a sequence of references starting from m.*

$$conf(r, R, m, M) = \begin{cases} true, & SM(r) = SM(m) \wedge \\ & \forall \langle r, r' \rangle \in R \colon \exists \langle m, m' \rangle \in M \colon \\ & conf(r', R, m', M) \\ false, & otherwise \end{cases} \tag{4.3}$$

Given an arbitrary reference metamodel, Definition 6 is potentially recursive. When performing conformance checking, this could lead to non-termination if circles exist in the reference metamodel. Consequently, reference metamodels have to be designed so that they do not contain circles. Then, the recursion terminates eventually and conformance can be checked automatically.

Conformance definitions similar to Definition 6 can be found in related work. Bucaioni et al. [39] determine the *conformance* between specific components of a specific architecture and an abstract component of a reference architecture. Here, a specific component *fully* conforms to a mapped abstract component if all mappings of its specific connectors to the abstract connectors of the corresponding abstract component are legal. The abstract and specific components described by Bucaioni et al. [39] are analog to metaclasses in the metamodels of the analyses and the abstract components to the metaclasses in our

reference metamodels. Additionally, the connectors of the components in the architectures are analog to the references of metaclasses in our metamodels. Seifermann et al. [300] determine *conformance* between two models of the same metamodel. Here, a model element in one model conforms to model elements in the other model if they can be reasonably mapped (e.g., by their name) and, amongst others, both have references to reasonably mapped model elements. Konersmann et al. [171] define *conformance* similar to Seifermann et al. [300] between reference models and specific models created with the same metamodel.

4.4.3. Process Steps

Here we present the process steps of the coupling process that are necessary to perform the parameterization described in Subsection 4.4.1. This process forms the basis for our coupling approaches presented in Chapter 5. However, to address different types of source code analyses and semantic relations, the process steps have to be specialized in the individual coupling approaches.

Our coupling process is realized by following process steps, illustrated in Figure 4.2:

Analyses Input Model Consistency. The *Analyses Input Model Consistency* step ($\mathcal{T}_{A\leftrightarrow C}(A, C^{-1})$) creates a *Source Code* which allows determining valid non-compliance with the source code analysis by establishing consistency with the *Annotated Architecture*.

Non-Compliance Analysis. The *Non-Compliance Analysis* step ($\mathcal{A}_C(C)$) performs the source code analysis to detect vulnerabilities in the implementation that can indicate non-compliance between the values of security characteristic resulting from the implementation and those specified in the *Annotated Architecture*.

Parameterization. The *Parameterization* step ($\odot(A, C, R_C)$) detects non-compliance and enriches the *Annotated Architecture* with the values of the security characteristic in the coupling scenario resulting from the implementation.

Architectural Analysis. The *Architectural Analysis* step ($\mathcal{A}_A(A)$) executes the architectural analysis with the *Annotated Architecture* that has been enriched in the *Parameterization* step to detect architectural vulnerabilities arising from non-compliant values of the security characteristic under investigation.

With these steps, we define the coupled analysis $\mathcal{A}_{coupled}(A, C^{-1})$ formally in Equation 4.4. In this equation, the coupling is realized by combining the output of the source code analysis $\mathcal{A}_C(C)$ with the architectural analysis input model A in the *Parameterization* step $\odot(A, C, \mathcal{A}_C(C))$ — i.e., the architectural model is parameterized with the results of the source code analysis. The resulting architectural model is then used as new input into the architectural analysis.

$$\begin{aligned} A' &= \mathcal{A}_{coupled}(A, C^{-1}) = \mathcal{A}_A(\odot(A, C, \mathcal{A}_C(C))) \\ \text{with} \quad C &= \mathcal{T}_{A\leftrightarrow C}(A, C^{-1}) \end{aligned} \tag{4.4}$$

In the descriptions of the processing steps in the following, we omit the models capturing the semantic relations. We omit these models because the information they capture and how they are used depends on the specific coupling approach — i.e., the analyses types to be coupled and the resulting semantic relations.

4.4.3.1. Analyses Input Model Consistency

$$\begin{aligned} &C = \mathcal{T}_{A \leftrightarrow C}(A, C^{-1}) \\ &\text{where} \quad \mathcal{T}_{A \leftrightarrow C}(A, C^{-1}) = \mathcal{CMPL}(\mathcal{CNST}(A, C^{-1})) \end{aligned} \tag{4.5}$$

The *Analyses Input Model Consistency* step $\mathcal{T}_{A \leftrightarrow C}(A, C^{-1})$ (Equation 4.5) is the first process step in the coupling. This step uses the current *Annotated Architecture A* and the *Source Code* before performing the evolution step C^{-1} as input. We denote the *Source Code* before performing the evolution step as C^{-1} to indicate that this model may have been modified in previous iterations of the coupling process. In initial development, we assume C^{-1} to be an empty model. The output of the *Analyses Input Model Consistency* step is an updated version of the *Source Code C* capturing elements which allow determining semantic relations to the *Annotated Architecture*.

The *Analyses Input Model Consistency* step is used to perform three goals: (1) Achieving a consistent representation of the system in the *Annotated Architecture* and *Source Code* regarding information necessary for the coupling, (2) Capturing relations between information in the *Annotated Architecture* and information in the *Source Code Analysis Result* via the *Source Code*, in the model to capture semantic relations, and (3) providing a *Source Code* that allows a comprehensive analysis of the implementation by the source code analysis to detect non-compliance with the *Annotated Architecture*. These goals are realized in the *Consistency* transformation $\mathcal{CNST}(A, C^{-1})$ and the *Completion* transformation $\mathcal{CMPL}(C)$.

The *Consistency* transformation addresses all goals by providing a *Source Code* that is consistent with the elements in the *Annotated Architecture* required for the coupling. This consistency is necessary to ensure that the same system is analyzed. Furthermore, the consistency of the *Source Code* with the *Annotated Architecture* enables the vulnerabilities detected in the *Source Code Analysis Result* to be related to the *Annotated Architecture*. This relation can be established because the *Source Code Analysis Result* is an abstraction of the *Source Code* [324]. If the system descriptions are inconsistent, we cannot assume that the source code analysis analyzes the same system as described in the *Annotated Architecture*. Consequently, the vulnerabilities in the *Source Code Analysis Result* cannot be reasonably related to the *Annotated Architecture*.

Due to the abstraction gap between architecture and implementation, we do not assume that consistency relations for all elements in the *Source Code* to the *Annotated Architecture* can be defined. For example, fields in classes are not represented in a component-based architecture like the PCM [275]. Additionally, behavior descriptions are likely to omit details used in the implementation (e.g., selection of encryption algorithms as in the approaches of Tuma et al. [338] or Seifermann et al. [300]). As a consequence, the *Source*

Code cannot be fully provided in the *Consistency* transformation. However, the *Source Code* provided by the *Analyses Input Model Consistency* step must contain all details necessary to perform a comprehensive analysis of the implementation.

For our running example, no implementation can be provided for the methods *buy*, *store* or *log*. Given a behavior description, omission of the field *privilegeSum* in the *Source Code* would prevent the source code analysis to detect the respective illegal information flow. Similarly, omission of the selection of a specific encryption algorithm in the *Source Code* would result in a source code analysis not being able to detect the usage of insecure algorithms. Consequently, the *Completion* transformation addresses goal (3) by adding elements to the *Source Code* which allow a comprehensive analysis of the implementation by the source code analysis.

Consistency transformation

$$\mathcal{CNST} : A \times C \rightarrow C \tag{4.6}$$

The *Consistency* transformation $\mathcal{CNST}$ (Equation 4.6) ensures the consistent representation of the system in the *Annotated Architecture* and *Source Code* based on correspondences between their elements. For this, the transformation uses the *Annotated Architecture A* and the *Source Code* before performing the evolution step C^{-1} as input and provides an *Incomplete Source Code C'*. C' only captures elements for which correspondences between the *Annotated Architecture* and *Source Code* can be defined. We assume consistency to be established either by generative approaches as proposed (e.g., used by Becker [24]) or consistency preservation approaches (e.g., the Vitruvius framework [167]).

The consistency transformation is created by the coupling expert to connect the different system abstractions in the *Annotated Architecture* and the *Source Code*. For this connection, the coupling expert interacts with the analysis architects of the analyses to be coupled to determine the correspondences between the elements of the *Annotated Architecture* and the *Source Code*.

The consistency must be established for elements in the *Annotated Architecture* that are necessary to perform the parameterization in the coupling. In our running example, for instance, the annotated parameters like *userData* in the service *buy* must have corresponding elements in the *Source Code* to allow the source code analysis to analyze the information flows affecting this parameter. In the *Consistency* transformation, we consider structural elements of the *Annotated Architecture* and *Source Code*, such as components, classes, interfaces, methods and connections between them. Examples for such correspondences are the mapping of components in the *Annotated Architecture* to classes in the *Source Code* or the mapping of provided interfaces of components to methods of classes [173, 181]. As we will see for the specification-based coupling approach in Section 5.1, the coupling may also require consistency for other information in the *Annotated Architecture* and *Source Code*, such as specifications of the security characteristic in the coupling scenario. We will discuss this need in the respective coupling approach.

During establishing the consistency, also mappings of elements in the *Annotated Architecture* to the model allowing to determine semantic relations between the *Annotated*

Architecture and *Source Code* have to be established. We discuss these mappings in the specific coupling approaches in Chapter 5 because they depend on the model capturing the semantic relations and the information available in the models of the analyses to be coupled.

Completion Transformation

$$\mathcal{CMPL}\colon C \rightarrow C \tag{4.7}$$

The *Completion* transformation $\mathcal{CMPL}$ (Equation 4.7) uses the *Incomplete Source Code* C' as input and provides a *Complete Source Code* C as output. Software engineers must provide implementation elements that do not have a corresponding architectural element in the *Annotated Architecture* but are relevant for the source code analysis. This task includes in the running example (see Chapter 3), amongst others, the implementation of methods and the provision of the field *privilegeSum*. The *Completion* transformation can also require adding security-related information to the *Source Code*, such as specifications for implementation elements which cannot be provided by the *Consistency* transformation. We discuss this need in the coupling approach involving the coupling of architectural analyses with specification-based source code analyses (Section 5.1).

4.4.3.2. Non-Compliance Analysis

$$R_C = \mathcal{A}_C(C) \tag{4.8}$$

The *Non-Compliance Analysis* step (Equation 4.8) uses the *Source Code* C as input, performs the source code analysis $\mathcal{A}_C(C)$ and provides the *Source Code Analysis Result* R_C as output. The source code analysis $\mathcal{A}_C(C)$ is performed in the *Source Code Analysis* transformation developed by the *analysis architect*. The applied analysis is selected by the coupling expert so that it can detect vulnerabilities affecting the security characteristic specified in the architectural analysis. Thus, the *Source Code Analysis* transformation identifies vulnerabilities that can result in non-compliance between the values of the security characteristic in the coupling scenario and those resulting from the implementation.

4.4.3.3. Parameterization

$$\begin{aligned} &A' = \odot(A, C, R_C) \\ &\text{where} \quad \odot\,(A, C, R_C) = \mathcal{I}\,(A, \mathcal{PMR}(C, R_C)) \end{aligned} \tag{4.9}$$

The *Parameterization* step $\odot(A, C, R_C)$ (Equation 4.9) parameterizes the architectural analysis based on the *Source Code Analysis Result*. We use the $\odot$ symbol according to the notation for combining analysis models used by Talcott et al. [323]. The parameterization step is performed by detecting non-compliance — indicated by vulnerabilities in the *Source Code Analysis Result*— and by modifying the annotations of the *Annotated Architecture* to represent the non-compliant values resulting from the implementation. For this purpose, the *Parameterization* step uses the *Source Code Analysis Result* R_C, the *Source Code* C, and

the *Annotated Architecture A* as input and provides a modified version of the *Annotated Architecture A′* as output.

This step performs the *Parameterization Model Resolution* transformation $\mathcal{PMR}(C, R_C)$ and the *Integration* transformation $\mathcal{I}(A, PM)$. The *Parameterization Model Resolution* transformation calculates the *Parameterization Model PM* from the *Source Code Analysis Result* and the *Source Code*. The *Parameterization Model* captures the information necessary to perform the parameterization (see Subsubsection 4.4.2.2). The *Integration* step uses the *Parameterization Model* and the *Annotated Architecture* to detect non-compliance, and to modify the *Annotated Architecture* accordingly. By using these transformations, we separate the parsing and interpretation of the *Source Code Analysis Result* (focused on the source code level) from detecting and handling non-compliance (i.e., bridging the gap between the architectural analysis and source code analysis) to separate the concerns of the coupling experts and the analysis architect.

Parameterization Model Resolution Transformation

$$\mathcal{PMR}\colon C \times R_C \rightarrow PM \tag{4.10}$$

The *Parameterization Model Resolution* transformation $\mathcal{PMR}(C, R_C)$ (Equation 4.10) uses the *Source Code Analysis Result* R_C and the *Source Code C* as input and provides the *Parameterization Model PM* as output. The *Parameterization Model Resolution* transformation extracts the information captured by the *Parameterization Model* from the *Source Code Analysis Result*. To this end, the *Parameterization Model Resolution* transformation has to parse the syntax in the *Source Code Analysis Result* to identify an implementation element that is affected by a vulnerability. In addition, the *Parameterization Model Resolution* transformation has to provide information that allows determining the values resulting from the implementation for the security characteristic in the coupling scenario.

The *Parameterization Model Resolution* transformation is defined by the analysis architect because they know the syntax and semantics of the information captured in the *Source Code Analysis Result* (e.g., the vulnerabilities found in the implementation or the implementation elements for which they are reported). Additionally, the analysis architect is able to provide information with the *Parameterization Model Resolution* transformation which is required by the *Parameterization Model* but missing in the *Source Code Analysis Result*. We exemplify this necessity in partial model **(g)** in Figure 3.2 of our running example where only the detected pattern identifier is presented in the *Source Code Analysis Result* but not the associated weakness. This approach avoids the challenge for the coupling expert to be aware of the syntax and semantics of the reported information to perform the parameterization.

Integration Transformation

$$\mathcal{I}\colon PM \times A \rightarrow A \tag{4.11}$$

The *Integration* transformation $\mathcal{I}(A, PM)$ (Equation 4.11) uses the *Annotated Architecture A* and the *Parameterization Model PM* as input and provides a modified version of the

Annotated Architecture A′ as output. This transformation is created by the *coupling expert* because it bridges the gap between information on the implementation level in the *Parameterization Model*, and information on the architectural level in the *Annotated Architecture*. The *Integration* transformation is responsible for detecting non-compliance and for integrating the values resulting from the implementation into the respective annotations in the *Annotated Architecture*. Thus, the *Integration* transformation has to calculate the values resulting from the implementation for the security characteristic in the coupling scenario based on the information captured in the *Parameterization Model*. Furthermore, the *Integration* transformation has to identify the annotations in the *Annotated Architecture* annotating the security characteristic to the architectural elements. With this information, the *Integration* transformation can detect non-compliance and modify the annotations in the *Annotated Architecture* accordingly.

In the coupling, we consider the *Annotated Architecture* as the prescriptive architectural definition of the system because it contains design decisions and information from the requirements phase [337]. Changing the original *Annotated Architecture* for detecting vulnerabilities arising from a non-compliant implementation regarding the specified security characteristic would result in losing this information. This is why the coupling creates a modified copy of the *Annotated Architecture* denoted with *A′*. This copy is intended for detecting vulnerabilities which arise from non-compliance to avoid this loss of information.

4.4.3.4. Architectural Analysis

$$R_A = \mathcal{A}_A(A') \tag{4.12}$$

The *Architectural Analysis* step (Equation 4.12) executes the architectural analysis $\mathcal{A}_A$ with the modified *Annotated Architecture A′* and produces the *Architectural Analysis Result* R_A as output. The purpose of this step is to detect the architectural vulnerabilities arising from the effects of non-compliance in the implementation regarding the security characteristic in the coupling scenario. This step is necessary to determine whether the architectural analysis reports new vulnerabilities after the parameterization.

However, a vulnerability in the architectural analysis only occurs if the modified *Annotated Architecture A′* violates a given security policy. Consequently, not every modification of the *Annotated Architecture* due to non-compliance will result in new architectural vulnerabilities. This scenario is illustrated in our running example (see Chapter 3) by the modification of the *Confidentiality Level* of the parameter *entry* of the *log* service. In this example, no new vulnerability arises despite the non-compliance between the value *high* resulting from the implementation, and the value *low* annotated to the corresponding parameter in the *Annotated Architecture*. This is why the coupling can also be used to determine which non-compliance in the implementation has an effect on the architectural security. However, currently, this requires manually relating the vulnerabilities in the *Architectural Analysis Result* back to the vulnerabilities in the *Source Code Analysis Result*. We discuss future work in Section 11.3 to support this task.

4.5. Limitations

Our coupling design is affected by several limitations. Future work to address these limitations is discussed in Section 11.3. We discuss six limitations in the following, enumerated as **L1** to **L6**.

Limitation 1: Manual Tasks Performed by the Coupling Expert The role *coupling expert* described in Section 4.2 aims to separate the responsibilities in the analysis coupling to allow specialization and reduce errors. Defining roles to clearly separate responsibilities and competencies is common practice in software engineering processes (e.g., Reussner et al. [275] guide the creation of component-based software systems by roles such as the software architect and component developer). Still, we see that all tasks assigned to the *coupling expert* that have to be performed manually are prone to errors. Errors in providing the transformation in the coupling process lead to incorrect results of the coupling. For example, the coupling expert can determine two elements to be semantically unrelated while they are actually semantically related as discussed in Subsection 4.3.1. However, omitting this separation of responsibilities and competencies would lead to other challenges, such as unclear responsibilities and errors that arise because of missing expertise. Thus, we see the impact of this limitation as acceptable.

Limitation 2: Lack of Tracing Architectural Vulnerabilities to Implementation Our coupling design focuses on detecting architectural vulnerabilities arising from non-compliance between the architecture and implementation. However, we do not provide an approach to relate the detected architectural vulnerabilities to the source of the non-compliance in the implementation. This challenge is further increased because the *Parameterization Model Resolution* transformation abstracts from the actual vulnerabilities. Consequently, the software engineers must determine *which* vulnerabilities in the implementation lead to the architectural vulnerabilities *manually*. This information is important for the software engineer to prioritize and address the vulnerabilities effectively. However, the *Parameterization Model* provides first indications about the source of non-compliance because it comprises the implementation elements related to the architectural elements affected by non-compliance. Furthermore, the effort for developing an approach for tracing the architectural vulnerabilities back to the implementation is lower because the transformation between the *Source Code Analysis Result* and the *Parameterization Model* is entirely allocated to a single role in the coupling process (i.e., the analysis architect).

Limitation 3: Focus on Static Source Code Analyses Currently, we focus only on static source code analyses. Using static source code analysis allows avoiding deployment of an insecure system. However, static analyses have limitations regarding the precision of their results due to over-approximation or under-approximation [4]. Dynamic analyses can provide more precise results regarding the actual behavior of the implementation. Approaches like fuzzing can also be used to detect vulnerabilities in the implementation

based on different inputs. This limits the applicability of our coupling approach to scenarios where static source code analyses are sufficient to detect non-compliance. However, as literature [4, 345] discusses, static analyses are more commonly used in practice than dynamic analyses. Thus, a focus on static analysis is a reasonable first step in investigating the coupling of architectural analyses and source code analyses to detect architectural vulnerabilities arising from a non-compliant implementation.

Limitation 4: Challenges in Determining Relations for Black-box Coupling We discussed the advantages of black-box coupling in Subsection 4.1.2. However, the black-box coupling also has drawbacks. The main drawback is that black-box coupling is not possible if the models of the analyses to be coupled do not have sufficient related information [70, 323]. This drawback is especially relevant as we detail in Subsection 4.3.1 that the insights into whether there are semantic relations (semantically related, semantically identical) or not (semantically unrelated) between two models can depend on the knowledge of the coupling expert. Thus, a coupling expert might determine a coupling to be impossible while another expert would detect a coupling. Finally, this limitation is also affected by the need for insights about the instances that are possible to be represented in the models due to the *model-level semantic refinement* (see Subsection 4.3.3).

Limitation 5: Detecting Non-Compliance only from Reported Vulnerabilities In our coupling design, we focus on detecting non-compliance based on vulnerabilities *reported* by the source code analysis. However, non-compliance can also arise due to the *absence* of implementation elements.

This limitation becomes evident in both coupling approaches presented later in Chapter 5. In the first coupling approach, non-compliance is only detected from *existing* information flows reported by the information flow analysis. However, information flows which *do not exist* can also result in non-compliance. As an example, assume that the parameter *userData* in the method *store* in our running example is annotated with the value *medium*. In this scenario, the architectural analysis performed in isolation of the implementation would report a vulnerability. Now assume that the information flow analysis reports no information flow to the *store* method. For example, this situation can arise because *userData* is strongly encrypted; resulting in the value *low* of *userData*. According to Definition 1, this situation also results in non-compliance because the value resulting from the implementation (i.e., *low*) cannot be mapped to the one specified in the design phase (i.e., *medium*). In particular, the architectural vulnerability reported by the architectural analysis in isolation would not occur when considering the value resulting from the implementation. However, we currently cannot detect this impact on architectural security because information flows are only reported by the source code analysis *if they exist*.

Similarly, the second coupling approach only calculates non-compliance arising from *existing* patterns reported in the *Source Code Analysis Result*. However, non-compliance can arise due to the *absence* of implementation elements. For illustrating this challenge, assume that *userData* in the *buy* method of our running example is sent with the *store*

method to the *DataStorage* without performing any encryption. In this scenario, non-compliance arises because *userData* is non-encrypted while the security characteristic for the communication link *Link1* specifies that data has to be encrypted when sending it over this link. This type of non-compliance cannot be detected in our pattern-search-based coupling approach because there is no pattern to be reported when encryption is not performed.

This limitation affects its completeness regarding all types of non-compliance. However, we argue that this limitation does not affect the correctness of the coupling approach as it can still detect architectural vulnerabilities arising from non-compliance for a significant subset of non-compliance that can arise.

Limitation 6: Reduced Generalizability Due to Exclusion of Analyses due to Omitting Behavior Description Omitting behavioral information in the architectural analysis for the coupling limits the applicable analyses. For example, design-level analyses such as SecDFD [338] or the DFA of Seifermann et al. [300] describe encryption in data flow diagrams or the SEFFs of the PCM [275] respectively. Our restriction to component-based architectural models without behavior description prevents our coupling design from being applied to such analyses. However, with our coupling design, we present a first step towards coupling architectural analyses with source code analyses to detect architectural vulnerabilities arising from non-compliance. Future work can address this limitation by extending the coupling design to support behavioral information in the architectural analyses.

4.6. Summary And Outlook

In this chapter, we presented the design of our analysis coupling as first contribution **C1** to answer **Research Question 1**. This coupling design addresses **Problem 1** by defining the properties of an analysis coupling suitable for detecting architectural vulnerabilities arising from non-compliant implementations. Furthermore, it defines the roles involved in the coupling, the semantics to consider when establishing couplings, and a comprehensive coupling process that operationalizes the coupling design.

By detailing this coupling design, we address the challenge identified by Durán et al. [71] that the composability of security properties — in our case the properties specified in the *Annotated Architecture* and those analyzed by the source code analyses — is insufficiently researched. Furthermore, the coupling design investigates when and how black-box coupling can be applied to bridge the gap between architectural and implementation-level security analyses.

Coupling Configuration We begin by specifying the *Coupling Configuration* in Section 4.1, which defines the properties of an analysis coupling suitable for detecting architectural vulnerabilities arising from non-compliant implementations. The configuration comprises

the *coupling goal*, the *coupling form* (what is composed), and the *composition type* (how the composition is realized).

The goal of the coupling is to detect architectural vulnerabilities arising from *specification-based non-compliance*. For this purpose, only *structural* information in the architectural analysis input model is used rather than behavioral descriptions. For the coupling form, we discuss the benefits of the selection of a black-box coupling over *grey-box* coupling and *white-box* coupling. The key advantage of black-box coupling is that it operates solely on the input and output *models* of the analyses and transformations between them. Consequently, this coupling avoids the need to modify the analyses themselves. This approach avoids the need to maintain one version of the analyses to use them in the coupling and one version to perform them in isolation of each other. The composition type employs *sequential analysis composition*, where results from the source code analysis are used to parameterize the architectural analysis. Specifically, the input model of the architectural analysis is enriched with values of security characteristics under investigation that result from the implementation rather than from the architectural specification. This approach defers the task of relating implementation-level vulnerabilities to architectural-level vulnerabilities to the architectural analysis itself.

Roles In Section 4.2, we define the roles involved in the coupling. We extend the capabilities and responsibilities of the roles *analysis architect* and *analysis user* defined by Koch [169] to the context of analysis coupling. Furthermore, we introduce the new role of *coupling expert*, who bridges the gap between independently developed analyses by defining relations between their input and output models. This separation of concerns enables analysis architects to develop independent analyses, while the coupling expert focuses specifically on establishing the connections between analyses.

Semantics Although the coupling operates at the syntactic level of input and output models, the coupling expert must consider the semantics of syntactic elements to establish meaningful couplings [324, 335]. In Section 4.3, we provide three types of semantic relations that guide the coupling expert: semantically related (syntactic elements map to overlapping semantic concepts), semantically identical (syntactic elements map to identical semantic concepts), and semantically unrelated (syntactic elements map to unrelated semantic concepts). To operationalize these semantic relations in the coupling process, we propose using *(meta)models* as syntactic representations of semantic relations. We also identify the challenge of *model-level semantic refinement*, which requires considering semantics at both the metamodel level and the model level to ensure valid couplings. This consideration is necessary because metaclasses can expose different semantics when instantiated.

Coupling Process Building upon the coupling configuration, roles, and semantic considerations, we propose a comprehensive *Coupling Process* in Section 4.4.

The coupling process performs two core tasks. First, it detects non-compliance between the architectural specification and the implementation regarding the security characteristic

under investigation. Second, it parameterizes the architectural analysis input model with the values of the security characteristic that result from the implementation based on the detected non-compliance. These tasks enable the architectural analysis to detect vulnerabilities based on the realized system rather than on assumptions from an earlier design phase.

To realize these tasks, the black-box coupling process utilizes five (meta)models: (1) the input model of the architectural analysis (*Annotated Architecture*), (2) the output model of the architectural analysis (*Architectural Analysis Result*), (3) the input model of the source code analysis (*Source Code*), (4) the output model of the source code analysis (*Source Code Analysis Result*), and (5) an intermediate model (*Parameterization Model*) that represents information necessary to perform the parameterization. The *Parameterization Model* addresses the challenge that the *Source Code Analysis Result* reports vulnerabilities rather than values of security characteristics, requiring interpretation to extract the information needed for parameterization.

We abstract from specific input and output metamodels of analyses, and from specific security characteristics using reference metamodels. These reference metamodels define the information necessary to perform coupling as reference metaclasses and references between them. The input and output metamodels of specific analyses must conform to these reference metamodels to be usable in the coupling process (see Definition 6). Consequently, the reference metamodels represent interfaces for the analysis coupling. This abstraction enables coupling approaches that are applicable to types of analyses and types of security characteristics rather than being tailored to specific analyses or specific security characteristics.

We define four process steps to achieve the coupling. Each step involves transformations between the models of the analyses and the *Parameterization Model*, consistent with the black-box coupling form. First, the *Analyses Input Model Consistency* step establishes consistency between the *Annotated Architecture* and the *Source Code* for elements relevant to the coupling and captures relations necessary for parameterization. Because it is unlikely that all information can be captured by consistency relations due to the abstraction gap, this step also includes completion of the *Source Code* to ensure it is comprehensively analyzable by the source code analysis. Secondly, the *Non-Compliance Analysis* step applies the source code analysis to detect vulnerabilities that may lead to non-compliance between the architectural specification and the implementation. Thirdly, the *Parameterization* step parameterizes the *Annotated Architecture* based on the vulnerabilities found in the *Non-Compliance Analysis* step. This involves deriving the *Parameterization Model* from the *Source Code Analysis Result* to capture the required information, then using this information to parameterize the *Annotated Architecture*, creating a parameterized architectural analysis input model. Fourthly, the *Architectural Analysis* step applies the architectural analysis to the parameterized model to detect architectural vulnerabilities arising from non-compliance in the implementation.

Limitations We conclude the presentation of our coupling design with a discussion of limitations in Section 4.5. We identify limitations regarding manual tasks assigned to the coupling expert, lack of tracing architectural vulnerabilities back to the implementation, focus on static source code analyses, and challenges in determining semantic relations in black-box coupling. These limitations inform future research directions to enhance the coupling approach.

Outlook Our coupling design provides the conceptual foundations for our second contribution **C2**: the instantiation of this design into specific coupling approaches. However, realizing a concrete coupling requires specialization for specific analysis types and security characteristics, because of great variations of the information available in the (meta)models. Consequently, also the realization of process steps has to address this information, especially the semantic relations between the (meta)models of the analyses. Therefore, the next chapter (Chapter 5) presents two coupling approaches that instantiate the proposed coupling design for specific types of source code analyses and security characteristics under investigation. Specifically, we present a coupling approach for coupling an architectural analysis with a static information flow analysis to detect architectural vulnerabilities arising from non-compliant information flows, and a coupling approach for coupling an architectural analysis with a static analysis to detect architectural vulnerabilities arising from a non-compliant implementation due to cryptographic errors. The feasibility of the coupling design and the effectiveness of these coupling approaches are then evaluated together in Chapter 8.

5. Coupling Approaches for Different Types of Source Code Analyses

Literature: This chapter is based on the following (co-) authored publications: [IEEE TSE 2025], [IEEE ICSA-C 2025], and [ACM MODELS-C 2024]

The coupling design in Chapter 4 provides the properties that coupling approaches must expose to couple an architectural security analysis with a static source code security analysis to detect architectural vulnerabilities arising from a non-compliant implementation. However, providing a single coupling approach for coupling all types of analyses is not feasible because of the variety of types of security characteristics, their value ranges, the security properties analyzed, and semantic relations between the (meta)models. Therefore, we present two coupling approaches as our contribution **C2** that *instantiate* the coupling design for two major types of source code analyses — namely *specification-based* source code analyses and *pattern-search-based* source code analyses (see Subsection 2.4.2). Furthermore, the coupling approaches address different types of security characteristics and different types of semantic relations between the (meta)models of the analyses, which results in different techniques to establish the coupling. Consequently, the coupling approaches illustrate two different approaches to establish the coupling.

The first coupling approach (described in Section 5.1) couples an architectural analysis specifying security characteristics of data which are influenced by information flows in the system with a *specification-based* source code analysis detecting information flow vulnerabilities. For the coupling, the semantically identical relation type between syntactic elements of the (meta)models of the analyses is exploited. We focus on defining necessary conditions that describe relations between the (meta)models of the analyses to establish the coupling. Focusing on these conditions is feasible because the semantically identical relation type reduces the complexity of operationalizing the calculation of relations between the syntactic elements.

The second coupling approach (described in Section 5.2) couples an architectural analysis specifying security characteristics describing the need for data encryption with a *pattern-search-based* source code analysis that detects patterns indicating encryption-related vulnerabilities. This coupling approach addresses the semantically related relation type between syntactic elements of the (meta)models of the analyses. The semantically related relation type does not provide direct correspondences between the (meta)models. Therefore, the second coupling approach focuses on techniques to identify related elements in the (meta)models of the analyses in the coupling. We discuss the semantic

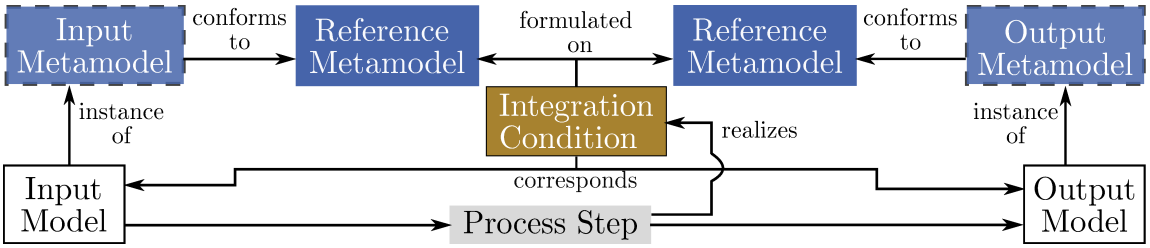


Figure 5.1.: Overview of the relation between models of the analyses, integration conditions and reference metamodels. See Figure 4.3 and Figure 4.2 for legend.

relations in more detail in the respective sections of the coupling approaches. For improved readability, we call the respective coupling approaches *specification-based coupling* and *pattern-search-based coupling* in the following.

The presentation of both coupling approaches is structured similarly in content for better comparability. We start by presenting the definition of *non-compliance* between the information in the *Annotated Architecture* and the implementation based on the analyzed security characteristic and analyses. This definition is specific to the type of source code analysis and the security characteristics addressed in each coupling approach. Next, we define the reference metamodels to capture information in the metamodels of the analyses necessary for the coupling. The non-compliance definition and the reference metamodels result in different types of semantic relations in the coupling. These relations influence the focus of the coupling approaches as discussed before. In the *specification-based coupling approach*, the semantically identical relation type allows focusing on the relation between (meta)models rather than on techniques to automatically identify related elements. This contrasts with the *pattern-search-based* coupling, which focuses on identifying related elements based on the semantically related relation type. Furthermore, we detail the coupling process by introducing information used to establish the coupling and by specializing the coupling process steps defined in Subsection 4.4.3. Introducing new processes and approaches involves additional efforts for their application. Thus, we continue with discussing the efforts arising in the realization of each coupling approach.

Due to the differences of the coupling approaches, their presentation can differ to account for information specific to each approach. We explicitly highlight the differences in the respective approaches.

After presenting both coupling approaches, we discuss limitations arising from the coupling design and the coupling approaches in Section 5.3. We conclude this chapter by summarizing both coupling approaches in Section 5.4.

5.1. Specification-based Coupling

Here, we describe our coupling approach for coupling architectural analyses with *specification-based* source code information flow analyses to investigate architectural vulnerabilities

arising from non-compliant values of security characteristics of data resulting from implemented information flows.

According to standards like the ISO 27000 series [140], confidentiality is one of the major security objectives. We select information flow in our coupling approach because confidentiality vulnerabilities tend to follow the data flow [300] and the control flow [20, 285]. Therefore, information flow vulnerabilities (i.e., the combination of data flow and control flows [20]) are especially important and should be considered throughout the development process. The importance of information flow is also reflected by the interest in scientific literature [19, 63, 80, 98, 159, 224, 283, 300].

Many static source code information flow analyses detect information flow vulnerabilities by using context-specific specifications of information flow related properties such as sources and sinks or the security classification of data communicated (e.g., [19, 80, 98, 100, 224, 283]). Consequently, we classify them as *specification-based* source code analyses (see Subsubsection 2.4.2.4).

Our specification-based coupling approach focuses on presenting the definition of *non-compliance*, the reference metamodels, Integration Conditions (ICs) which define necessary but not sufficient conditions to establish the coupling and the specialization of the coupling process. The interplay of the metamodels of the analyses, their models, ICs, reference metamodels and the coupling process in our specification-based coupling approach is illustrated in Figure 5.1.

The presentation of this information in the *specification-based* coupling approach is structured similarly to the description in Chapter 5. We first define *non-compliance* between the *Annotated Architecture* and information flows in the implementation in Subsection 5.1.1. We then abstract from specific analyses and their (meta)models in this coupling approach by defining the information required for the coupling in form of reference metamodels presented in Subsection 5.1.2.

However, black-box coupling is constrained by the semantic mappings between the (meta)models of the analyses [323]. To support the coupling expert in establishing the coupling, we define ICs in Subsection 5.1.3 that describe which formulate necessary but not sufficient conditions to establish the coupling. Necessary but not sufficient means, that these conditions must hold to establish a coupling, but the coupling is not guaranteed when they hold. Note that the ICs are formulated specifically for coupling architectural analyses and specification-based information flow source code analyses which we address in this coupling approach. We do not claim generalizability for the ICs to other security characteristics or types of analyses, but discuss this as part of our limitations in Section 5.3 and future work in Section 11.3. The ICs exploit the semantically identical relation type between elements of the (meta)models of the analyses in the coupling. This relation type reduces the complexity of mechanisms in the coupling process to (1) capture correspondences in transformations for elements in the models of the analyses related as required by the ICs, and (2) looking up the corresponding elements for the parameterization. These mechanisms can be assumed to have low complexity once the relations between the elements are defined. This allows us to focus on the ICs rather than to provide mechanisms

to calculate these relations in the coupling process. The description of the ICs deviates from the general structure of the coupling approaches presented in Chapter 5.

Based on the details provided by the reference metamodels and ICs, we specialize the coupling process in Subsection 5.1.4. Finally, we discuss the efforts arising in realizing the coupling in Subsection 5.1.5.

5.1.1. Defining Non-Compliance for the Specification-Based Coupling

In the specification-based coupling approach, we define non-compliance between the values of security characteristics of data specified in the *Annotated Architecture* and the values resulting from the information flows in the implementation. Accordingly, we restrict the coupling to security characteristics of data affected by information flows in the system.

An information flow is defined as the flow of information from one system element (*source*) to another system element (*sink*) [63]. A reported information flow indicates that the values of the security characteristics of data specified for the source are propagated to the sink. We refer to the values of security characteristics of data from the source that are propagated to the sink as the *values resulting from the implementation* for this sink. In our running example, the source code analysis reports a flow from the *high*-classified parameter *userData* in the *buy* method to the *low*-classified parameter *userData* in the *store* method (illustrated in partial model (f) in Figure 3.2). This information flow can be interpreted so that the parameter *userData* in the *store* method now contains data classified as *high*.

Applying Definition 1, we define non-compliance to arise when the implemented information flows result in values of security characteristics of data for a sink implementation element that cannot be mapped to the values specified for a corresponding architectural element in the *Annotated Architecture*. In our running example, the parameter *userData* in the *store* method and the parameter *userData* in the *store* service correspond to each other. In this case, non-compliance arises because the value *high* resulting from the implementation for the *userData* in the *store* method cannot be mapped to the value *low* specified for the corresponding parameter in the *Annotated Architecture*.

5.1.2. Reference Metamodels for Metamodels of Analyses in the Coupling

We abstract from specific analyses by describing reference metamodels for the metamodels of the analyses involved in the coupling (see Subsubsection 4.4.2.3). The reference metamodels define information necessary for the coupling by using metaclasses and references between these metaclasses. An analysis can only be used in the coupling if it conforms to the reference metamodels (again, see Subsubsection 4.4.2.3).

In our specification-based coupling approach, we focus on source code information flow analyses that require specifications in the *Annotated Source Code* (e.g., [16, 19, 98, 100, 224,

Reference Metamodels Sets	
Δ	System Elements
S	Security Characteristics
An	Annotations
CFG	Configurations
Pol	Security Policies
RE	Result Entry
REE	Result Entry Elements
PE	Parameterization Entries
(Meta-)model elements	
$\delta \in \Delta$	System Element
$\alpha \in An$	Annotation
$s \in S$	Security Characteristic
$cfg \in CFG$	Configuration
$pol \in Pol$	Security Policy
$r_k \in RE$	k-th Result Entry
$ree_k^i \in REE$	i-th Result Entry Element of r_k
$pe \in PE$	Parameterization Entry

Table 5.1.: Sets and Elements in the Reference Metamodels for the Metamodels of Analyses in the Specification-Based Coupling Approach.

283]). These specifications can be provided in different syntactic constructs which we subsume under the term *annotation* (see Subsubsection 2.3.2.2). Hence, we refer to the input model of the source code analysis as *Annotated Source Code* (rather than *Source Code*) to emphasize the focus on the annotations.

We detail a reference metamodel for the *Annotated Architecture* and the *Annotated Source Code* in Subsubsection 5.1.2.1, for the *Source Code Analysis Result* in Subsubsection 5.1.2.2 and for the *Parameterization Model* in Subsubsection 5.1.2.3. We omit a reference metamodel for the *Architectural Analysis Result* because it is not involved in the actual coupling. Table 5.1 lists all elements in the reference metamodels and their formula symbols.

5.1.2.1. Reference Metamodel: Input Metamodels

The reference metamodel for the *Annotated Architecture* and *Annotated Source Code* is presented in Figure 5.2. We partition the reference metamodel into *System Elements*, describing the system, *Security Specification Elements* for the representation of the specification, and *Configuration Elements* used for configuration of the analysis.

System Elements A *system element* $\delta \in \Delta$ in a DSL Δ_A or programming language Δ_C describes structural information or behavioral information of the system. In component-based architectures like the PCM [275] or the UML [235], system elements which represent the structure include components, interfaces, or connectors. The SEFF in the PCM is

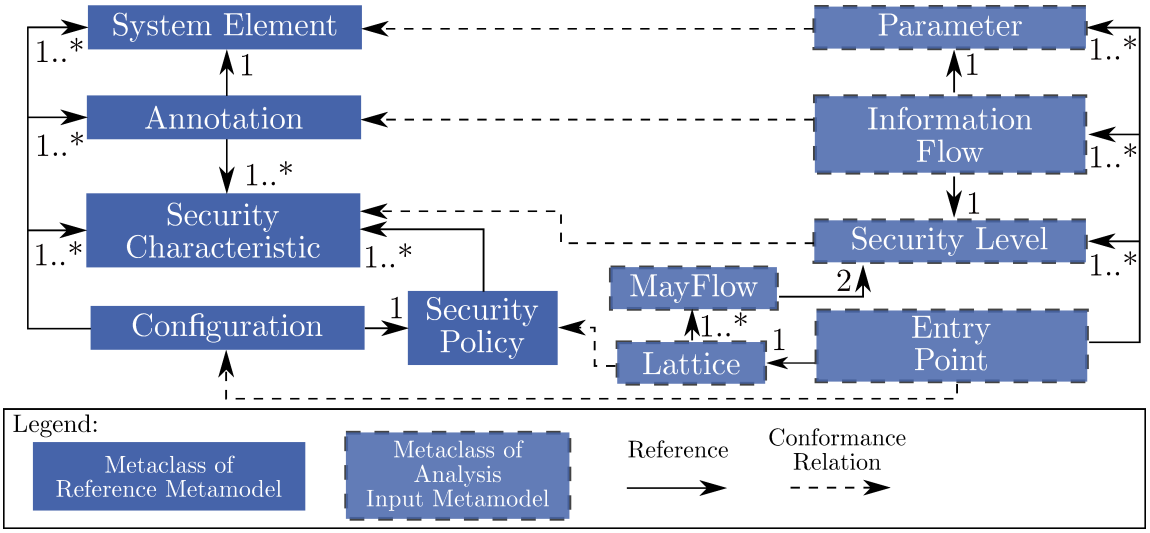


Figure 5.2.: Metaclasses and References of the reference metamodel for *Annotated Architecture* and *Annotated Source Code* (left), and metaclasses and references for *JOANA* in the running example in Section 3.2 (right). *EntryPoint* from actual *JOANA* [98] and conformance relations included. Source code for configuration abstracted by only referencing *Parameter*. Metaclasses of reference metamodel and metaclasses of running example illustrated conforming to Figure 4.3.

an example of behavioral descriptions that include elements such as internal actions or external calls. Programming languages like Java provide system elements that represent structural constructs such as classes, methods, and fields or behavioral constructs such as statements in methods. We do not consider behavioral elements in the *Annotated Architecture* for the coupling in this dissertation (see Subsection 4.1.1). Thus, in the running example, relevant system elements include parameters or resources in the *Annotated Architecture*.

Security Specification Elements The security specification includes the metaclasses *Security Characteristic* s and *Annotation* α to describe security-relevant properties of *System Elements* δ.

Security Characteristic. We expect the *Annotated Architecture* and the *Annotated Source Code* to contain one or more *Security Characteristics* $s \in S$. A security characteristic s describes security-related properties of *System Elements* δ (see Section 2.4). We assume the *type* of a security characteristic to be captured by a metaclass and its *values* to be represented by instances of this metaclass. In our running example (Chapter 3), *confidentiality level* is the *type* of the security characteristic in the *Annotated Architecture* and the values *high* and *low* are correspondingly named instances. We require both analyses in the coupling to use security characteristics of data that are affected by the information flow. Such security characteristics of data can be found in various architectural analyses (e.g., [96, 150, 172, 300, 334, 338]) and various source code analyses (e.g., [98, 224]). With this requirement, we establish a semantically identical relation between the *Annotated Architecture* and the *Annotated Source Code*.

For our coupling approach, the analyses can either predefine the semantics and values of a security characteristic for the analysis, or they allow the user to define them. The analysis of Gerking and Schubert [96] is an example where a metaclass *sensitivity* is defined with the predefined values *secret*, *public* and *neutral*. In contrast, the metaclass *Level* in *JOANA* [98] allows the specification of various security classifications of data (e.g, confidentiality levels or roles) with values defined by the analysis user.

Annotation. The *Annotated Architecture* and the *Annotated Source Code* contain one or more *Annotations* $\alpha \in An$ that enable the specification of *Security Characteristics* for *System Elements*. An *Annotation* references one or more *Security Characteristic* and one *System Element* as expressed in Equation 5.1.

$$\forall \alpha \in An \colon \exists \delta \in \Delta, \exists s \in S \colon \langle \alpha, \delta \rangle \wedge \langle \alpha, s \rangle \tag{5.1}$$

We assume the *Security Characteristics* referenced by an *Annotation* to belong to the same semantic domain, such as information flow security. Similarly, we expect different *Annotations* belonging to the same semantic domain to annotate the same values of *Security Characteristics* to a *System Element*. For example, *JOANA* [98] provides the *Annotations Source* and *Sink* which both reference the security characteristic *Level*. If one annotation specifies the value *low* for a parameter, while another annotation specifies the value *high* for the same parameter in the same execution of the analysis, it would be unclear whether the parameter is expected to contain *high* or *low* data. Based on this assumption, we address the challenge in selecting annotations in the parameterization process described in Subsection 4.4.1. If all *Annotations* which belong to the same semantic domain annotate the same value of a *Security Characteristic* for a *System Element*, it suffices to select all *Annotations* annotating the same *Security Characteristic* to perform the parameterization. In particular, non-compliance detected in one annotation, implies that *all* annotations annotating the same security characteristic are affected by non-compliance. This assumption reduces the complexity of selecting the correct annotations to be used in the parameterization to identifying the security characteristic they annotate.

Analyses exist that provide annotations without explicitly defining security characteristics (e.g., [19, 96, 100, 338]). Thus, the semantics of the annotations must be clear to understand which value of security characteristic is implicitly represented by the annotation. The challenge arises for the coupling that information flow analyses can investigate the security objectives *confidentiality* or *integrity* [20, 63]. In confidentiality analysis, it is analyzed whether an information flow exists from confidential information to non-confidential information. For such analyses, sources can be interpreted to contain confidential information and sinks to contain non-confidential information. In contrast, integrity analysis investigates whether untrusted information influences trusted information. Therefore, sources can be interpreted to contain untrusted information and sinks to contain trusted information. Confidentiality and trust can both be interpreted as security characteristics with the values *high* and *low* [63]. In cases where an analysis targets only a single security objective or the security objective is explicitly stated (e.g., in SecDFD [338]), the semantics of the annotations is unambiguous. Consequently, the security characteristic and its values can be inferred from the sources and sinks (e.g., *high* for sources and *low* for

sinks in confidentiality analysis). However, there are analyses like CodeQL [100, 214] and FlowDroid [19] that do not explicitly state the security objective to be addressed. Here, the semantics of the annotations depends on the special use case which cannot be inferred from the metamodel of the analysis.

Analysis Configuration Elements The analysis configuration metaclasses comprise the *Security Policy pol ∈ Pol* and *Configuration cfg ∈ CFG*. They define the input (i.e., system and specification) used in performing the analysis and when the analysis must report a vulnerability.

Security Policy. Specification-based analyses detect vulnerabilities based on a *Security Policy pol ∈ Pol*. The *Security Policy* references one or more *Security Characteristics* (as specified in Equation 5.2) for describing the allowed or unallowed relations between the security characteristics.

$$\forall pol \in Pol\colon \exists s \in S\colon \langle pol, s\rangle \tag{5.2}$$

Our running example shows two security policies. Partial model (b) of Figure 3.1 illustrates the security policy of the architectural analysis defining that confidential data (*high* or *medium*) must not be processed on resources that are deployed outside the EU (*Non-EU*). In contrast, partial model (e) of Figure 3.2 illustrates the security policy of the source code analysis defining that only information assigned a lower confidentiality level is allowed to flow to program locations expected to contain information with a higher confidentiality level.

We found that there are two kinds of security policies: *configurable Security Policies* and *integrated Security Policies*. Analyses can allow the specification of the relations between the values of security characteristics in security policies (e.g., [98, 150, 159, 300]) which allows the analysis user to *configure* when a vulnerability is reported. An example is the lattice provided by *JOANA* [98] that enables the user to define allowed information flows between confidentiality levels. Seifermann et al. [300] enables the analysis user to specify which flow between values of security characteristics is allowed. However, analyses can also use an *integrated* security policy that uses predefined rules to define the relation between the values of security characteristics (e.g., [19, 96, 172, 338]). For example, Gerking and Schubert [96] define rules that define which flows are allowed between their provided sensitivity levels secret, public and neutral.

The possible values of *Security Characteristics* occurring in the *Source Code Analysis Result* are based on the values described in the *Security Policy*. For example, if the security policy does not consider the value *medium* of the security characteristic *confidentiality level* in our running example, the analysis cannot report vulnerabilities involving this value.

Configuration. The models used when performing an analysis must be identifiable to integrate information from the *Source Code Analysis Result* into the *Annotated Architecture* in the *Parameterization* step of the coupling process. A *Configuration cfg ∈ CFG* captures information required to perform an analysis. This information includes the *System Elements* to be analyzed Δ, the available values of the *Security Characteristics S*, their *Annotations*

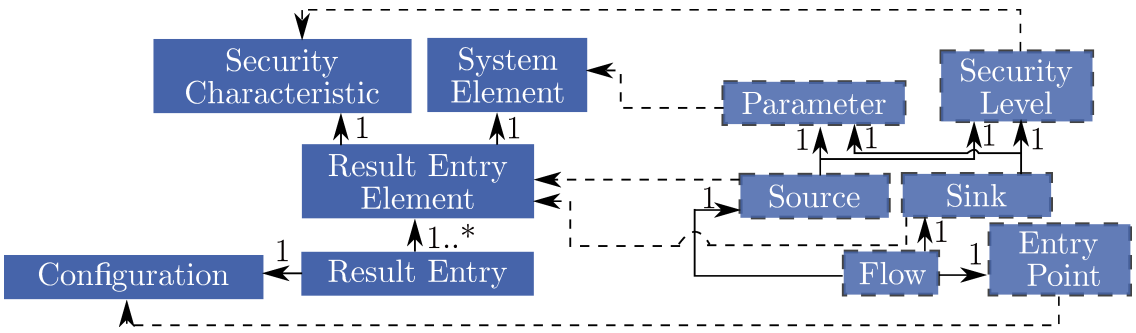


Figure 5.3.: Metaclasses and references of the reference metamodel for *Source Code Analysis Result* (left), and metaclasses and references for *JOANA* in the running example in Section 3.2 (right). Please see Figure 5.2 for the legend.

An to *System Elements* δ, and the applied *Security Policy* $pol \in Pol$. Thus, we express a configuration formally in Equation 5.3.

$$\begin{aligned} cfg \in CFG\colon \exists \alpha \in An, \exists s \in S, \exists \delta \in \Delta, \exists! pol \in Pol\colon \\ \langle cfg, s\rangle \wedge \langle cfg, \delta\rangle \wedge \langle cfg, \alpha\rangle \wedge \langle cfg, pol\rangle \end{aligned} \tag{5.3}$$

In our running example, the configuration in *JOANA* [98] defines the source code to be analyzed (e.g., the implementation of the classes *OnlineShop*, *DataStorage* and *Logger*), the available security levels (e.g., *high*, *medium* and *low*), the applied annotations, and the applied security policy. This information enables detecting the vulnerabilities arising in the system. The *Configuration* references a *Security Policy* because the analysis results represent policy violations.

A *Configuration* references one or more *Annotations* used in the analysis. The requirement that all *Annotations* of the same semantic domain annotating the same values of the *Security Characteristics* is expected to hold only in a *single Configuration*. Thus, we allow annotations of the same domain to annotate different values of a *Security Characteristic* for the same *System Element* in *different configurations*. Allowing annotations of different values of the same security characteristic in different configurations enables what-if analysis or supporting constraints in analysis algorithms. For example, *JOANA* [98] does not allow the annotation of a single *System Element* as *Source* and *Sink* in the same configuration.

Several analyses are configured by the input of the analysis without a dedicated configuration metaclass (e.g., [19, 150, 300, 338, 348]). Other analyses can have a configuration metaclass identifying the *Annotations*, *Security Characteristics* and *Policies*, but the system to be analyzed is provided as dedicated input for the analysis (e.g., [19, 98, 100]). We account for the configuration without a dedicated metaclass or a mixture of both by treating the set of all input models for the analyses as *Configuration*. We call a *Configuration explicit* if a dedicated metaclass exists and *implicit* otherwise.

5.1.2.2. Reference Metamodel: Output Metamodels

To perform the coupling, we assume the *Source Code Analysis Result* to report counter-examples containing elements of the *Annotated Source Code* rather than quantitative values or binary yes-or-no statements (i.e., an information flow exists that violates the security policy). Only a *Source Code Analysis Result* containing elements of the *Annotated Source Code* allows to establish relations to the *Annotated Architecture* and to calculate the *values of the security characteristics resulting from the implementation.*

The reference metamodel for the *Source Code Analysis Result* defines metaclasses necessary in the counter-examples (depicted in Figure 5.3). A common structure for the *Source Code Analysis Result* in information flow analysis is to represent the source and sink of an information flow that cause a vulnerability (e.g., in [19, 82, 98, 100]). In the running example, an illegal flow is represented by providing the field *privilegeSum* with the confidentiality level *medium* as source and the parameter *userData* with the confidentiality level *low* as sink (illustrated in partial model **(f)** in Figure 3.2).

We define the reference metamodel to comprise the metaclass *Result Entry* $r_k \in RE$ which represents a single counter-example and contains one or more *Result Entry Elements* $ree_k^i \in REE$. Calculating the *Annotations* that specify the *Security Characteristics* of the coupling scenario in the *Annotated Architecture* from the information in the *Source Code Analysis Result* is only possible if we can unambiguously identify all annotations. With our assumption that *Annotations* of the same semantic domain must annotate the same values of *Security Characteristics* for a *System Element* (see paragraph 5.1.2.1), a combination of *System Element*, *Security Characteristic*, and *Configuration* suffices to resolve annotations in the input model. Consequently, a *Result Entry Element* ree_k^i references a *System Element* $\delta \in \Delta$ and a *Security Characteristic* $s \in S$ as expressed in Equation 5.4.

$$\forall ree_k^i \in REE\colon \exists \delta \in \Delta, \exists s \in S\colon \langle ree_k^i, \delta \rangle \wedge \langle ree_k^i, s \rangle \tag{5.4}$$

In addition, the *Result Entry* references a *Configuration.* This relation expresses that the counter-example expressed by the *Result Entry* is calculated using this configuration. This information allows to relate the found counter-examples to the input used by the analysis and enables the resolution of the *Annotations.*

Consequently, we describe a *Result Entry* $r_k \in RE$ formally in Equation 5.5.

$$r_k \in RE\colon \exists! cfg \in CFG, \exists ree_k^i \in REE\colon \langle r_k, cfg \rangle \wedge \langle r_k, ree_k^i \rangle \tag{5.5}$$

The metaclasses in the *Result Entries* and *Result Entry Elements* also exist in the reference metamodel of input metamodels because the *Source Code Analysis Result* is an abstraction of the *Annotated Source Code* [324]. Hence, the *Source Code Analysis Result* can be realized by metamodel elements of its own, by references to elements of the *Annotated Source Code*, or by identification of the elements of the *Annotated Source Code* (e.g., by using the name of a field and the fully qualified path of its class).

5.1.2.3. Reference Metamodel: Parameterization Model

The reference metamodel for the *Parameterization Model* is similar to the one for the *Source Code Analysis Result* but abstracts the counter-examples and the interpretation performed by the *Parameterization Model Resolution* transformation (see Subsubsection 4.4.2.2). In the specification-based coupling approach, we expect the analysis architect to provide the *values of the security characteristic in the coupling scenario resulting from the implementation* for an implementation element already in the *Parameterization Model*. This is possible (1) because the coupling process exploits the semantically identical relation type between the *Annotated Architecture* and the *Annotated Source Code*, and (2) because of how specification-based source code analyses calculate their vulnerabilities (further detailed in Subsection 5.1.4).

This information is represented in the reference metamodel for the *Parameterization Model* by the metaclass *Parameterization Entry*. The *Parameterization Entry* metaclass references a *Security Characteristic*, a *System Element*, and a *Configuration* as formally described in Equation 5.6. Again, this information suffices to identify the annotations and the security characteristic to be modified in the *Parameterization* process step.

$$\forall pe \in PE\colon \exists s \in S, \exists \delta \in \Delta, \exists cfg \in CFG\colon \langle pe, s\rangle \wedge \langle pe, \delta\rangle \wedge \langle pe, cfg\rangle \tag{5.6}$$

5.1.3. Integration Conditions

Addressing the semantically identical relation type in the specification-based coupling approach shifts the focus from mechanisms used to calculate the relations between the model elements to determining these relations for the transformations. We support coupling experts in this task by defining 10 Integration Conditions (ICs) as illustrated in Figure 5.4. The ICs formulate necessary but not sufficient relations that must hold to establish the coupling. Every IC is formulated for a specific process step in the coupling process presented in Subsection 4.4.3.

In the *Parameterization* step, we have to retrieve annotations annotating the security characteristic of data of the coupling scenario in the *Annotated Architecture* and replace their specified values with those resulting from the implementation. Consequently, the *Integration* transformation must receive a *Parameterization Model* containing information that allows (1) resolving the annotations affected by the non-compliance and (2) integrating values resulting from the implementation into the *Annotated Architecture* that the architectural analysis can process. As described in Subsubsection 5.1.2.2, we can unambiguously identify an *Annotation* by a *Security Characteristic*, *System Element*, and a *Configuration*. Thus, we formulate the ICs to ensure resolution capabilities between the *Parameterization Model* and *Annotated Architecture* for these elements to ensure that the *Annotated Architecture* can be parameterized with the values resulting from the implementation. For this, the ICs ensure that the *Annotated Architecture* contains metaclasses and instances which correspond to elements in the *Parameterization Model* (e.g., the *Level high* and the

SecurityCharacteristic high in the running example). A *coupling expert* can also use the ICs to elaborate correspondences for the transformations in the coupling process.

The ICs are classified into two categories: *correspondence conditions* and *constraint conditions*.

> **Correspondence Conditions.** A correspondence condition induces correspondences between metaclasses (and their instances) in two (meta)models in a process step. These conditions are described so that correspondence for at least one metaclass and at least one of their instances is required. As already discussed before, we base this coupling approach on the semantically identical relation between the (meta)model elements of the analyses in the coupling. Consequently, we expect the elements corresponded by *correspondence conditions* to be semantically identical. We denote correspondence conditions in the following with *(T)* to distinguish them from the *constraint conditions* and because they can be also interpreted to introduce *traces* [147] between the elements of the (meta)models involved in the IC.
>
> **Constraint Conditions.** A constraint condition constrains at least one or all metaclasses (and at least one or all of their instances) that conform to the metaclass of the reference metamodels to reference other metaclasses (and their instances) for which a *correspondence condition* exists. With constraints for at least one metaclass (and at least one of its instances) and constraints for all metaclasses (and all of its instances), we ensure that only as many elements of the (meta)models as necessary are addressed for the coupling. We denote constraint conditions in the following with *(C)*.

The ICs are presented in dedicated boxes for better readability in the following. These boxes are colored according to their type as illustrated in Figure 5.4: green for correspondence conditions and purple for constraint conditions.

Formulating the ICs on the metamodel level does not suffice, because the validity of a resolution may depend on the specific models due to *model-level semantic refinements* (see Section 4.3). Furthermore, if the values of the security characteristic under investigation that result from the implementation (calculated in the *Parameterization Model Resolution* step; further detailed in Subsection 5.1.4) cannot be resolved to the *Annotated Architecture*, the coupling cannot be established. Thus, we specify the ICs to hold on the metamodel and model level.

Some correspondence conditions describe correspondences between different metaclasses of the reference metamodels to increase conciseness in this thesis. Providing separate ICs for every single metaclass would be more precise. However, this approach would result in 15 ICs on the metamodel level in contrast to the 10 ICs presented in this section on the metamodel level. In addition, presenting all ICs on the metamodel and model level in this section would result in 20 ICs instead of 10 without the refinements and 30 ICs including the refinements. Presenting this amount of ICs would reduce the readability of this section significantly. Thus, we decided to present **IC1** as one example to illustrate an IC on the metamodel and model level and **IC2** as an example to illustrate a refinement of an IC into separate conditions. In the first example, we denote the metamodel level with

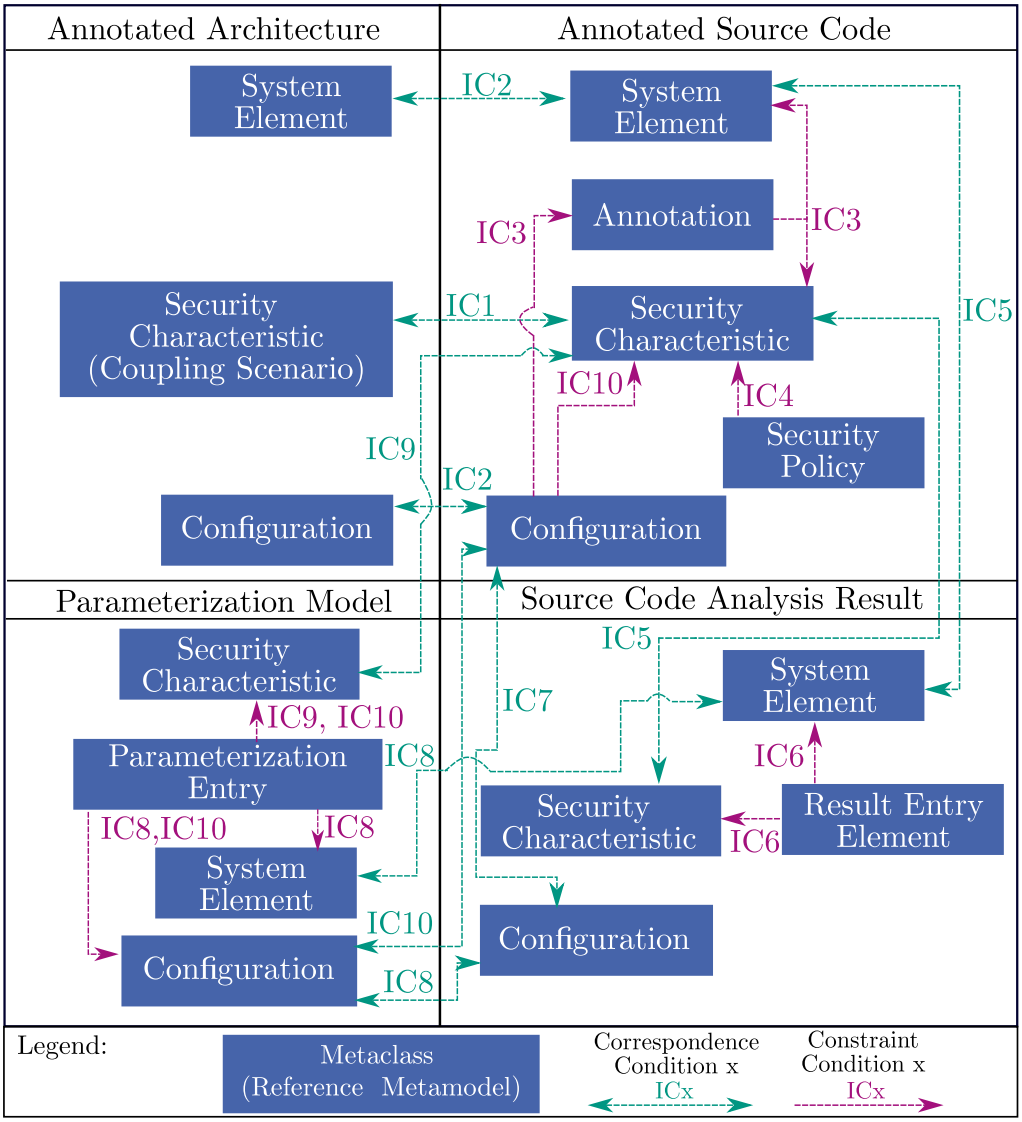


Figure 5.4.: Integration Conditions for the Specification-Based Coupling Approach described for the Metaclasses of the Reference Metamodels. Metaclasses not affected by any IC and references between metaclasses are omitted for better readability. Please refer to Subsection 5.1.2 for all metaclasses and references of the Reference Metamodels.

(M) and the model level (i.e., instance level) with *(I)*. The refinements are denoted with $X.Y$ where X describes the original IC and Y consecutively numbers the refinements. We provide all refinements of all ICs where applicable in our replication package [270]. We will provide the remaining ICs in this section only on the metamodel level and without the refinements for better comprehensibility.

5.1.3.1. Integration Conditions: Analyses Input Model Consistency

The *Annotated Source Code* connects the *Source Code Analysis Result* and the *Parameterization Model* to the *Annotated Architecture*. Thus, we have to establish the ability to resolve

the security characteristic in the coupling scenario and the respective annotations in the *Annotated Architecture* from the *Annotated Source Code*.

The instances of the metaclass describing the security characteristic of data in the coupling scenario s_A^{cs} form the set S_A^{cs}. Let $\underline{s_A^{cs}}$ denote instances of the metaclass s_A^{cs} (forming the set $\underline{S_A^{cs}}$). For our coupling approach to work, we also have to assume that the architectural analysis uses at least one annotation $\alpha_A^{cs} \in An_A^{cs} \subset An_A$ that annotates the security characteristic of data s_A^{cs} in the coupling scenario to system elements (formally described in Equation 5.7).

$$\begin{gathered}(An_A^{cs} \neq \emptyset) \\ \text{where} \\ An_A^{cs} = \{\alpha_A \in An_A \mid \exists s_A \in S_A : \langle \alpha_A, s_A \rangle \wedge s_A = s_A^{cs}\}\end{gathered} \tag{5.7}$$

With this assumption, we can start formulating the ICs. We first require with **IC1** that the security characteristic of the coupling scenario in the *Annotated Architecture* must be resolvable from the *Annotated Source Code*. As the *Parameterization Model* contains security characteristics (and their values) occurring in the *Annotated Source Code*, this IC ensures their resolution to the *Annotated Architecture*. In the running example, this IC is fulfilled because the metaclass *confidentiality level* and the corresponding values *high*, *medium* and *low* from the source code analysis can be resolved to the confidentiality levels in the architectural model.

IC1(T)(M): A *Security Characteristic* s_C must exist in the *Annotated Source Code* C corresponding to the *Security Characteristic* of the coupling scenario s_A^{cs} in the *Annotated Architecture* A.

$$\begin{gathered}S_C^{cs} \neq \emptyset \\ \text{where} \\ S_C^{cs} = \{s_C \in S_C \mid s_C \leftrightarrow s_A^{cs}\}\end{gathered}$$

IC1(T)(I): A value $\underline{s_C}$ of a *Security Characteristic* s_C in the *Annotated Source Code* C must exist, corresponding to an instance $\underline{s_A^{cs}}$ to the *Security Characteristic* of the coupling scenario s_A^{cs} in the *Annotated Architecture* A.

$$\begin{gathered}\underline{S_C^{cs}} \neq \emptyset \\ \text{where} \\ \underline{S_C^{cs}} = \{\underline{s_C} \in \underline{S_C} \mid \underline{s_C} \leftrightarrow \underline{s_A^{cs}}\}\end{gathered}$$

To resolve annotations annotating the security characteristic of the coupling scenario in the *Annotated Architecture*, we also require the annotated system elements and the

configurations in which the annotations are used (see Subsubsection 5.1.2.2). Resolving the configuration is essential as the coupling investigates non-compliance regarding a specific system and specification (which contains the annotations of the security characteristics under investigation). Consequently, the configuration used in the *Annotated Architecture* must be resolvable from the *Annotated Source Code* to consider only the annotations applied for analysis. Therefore, we require with **IC2** resolution capabilities for the system elements and configuration involved in annotating the security characteristics in the coupling scenario.

IC2(T): There must exist

- *System Elements* δ_C of the *Annotated Source Code C* which correspond to *System Elements* δ_A of the *Annotated Architecture A* that are annotated with the *Security Characteristic* s_C^{CS} of the coupling scenario by at least one *Annotation* α_C and
- *Configurations* cfg_C of the *Annotated Source Code C* which correspond to *Configurations* cfg_A in the *Annotated Architecture A* that use at least one *Annotation* α_C, annotating the *Security Characteristic* s_C^{CS} in the coupling scenario.

$$\Delta_C^{cs} \neq \emptyset \wedge CFG_C^{cs} \neq \emptyset$$

where

$$\Delta_C^{cs} = \{\delta_C \in \Delta_C \mid \exists \delta_A \in \Delta_A \colon \exists \alpha_A^{cs} \in An_A^{cs} \colon \langle \alpha_A^{cs}, \delta_A \rangle \wedge \delta_C \leftrightarrow \delta_A\}$$

$$CFG_C^{cs} = \{cfg_C \in CFG_C \mid \exists cfg_A \in CFG_A \colon \exists \alpha_A^{cs} \in An_A^{cs} \colon \langle cfg_A, \alpha_A^{cs} \rangle \wedge cfg_C \leftrightarrow cfg_A\}$$

The correspondences between δ_C and δ_A and correspondences between cfg_C and cfg_A required by **IC2** do not have any dependencies between those elements. Thus, we can refine the correspondence condition **IC2** into two separate correspondence conditions **IC2.1** and **IC2.2**.

IC2.1(T): There must exist *System Elements* δ_C of the *Annotated Source Code C* which correspond to *System Elements* δ_A of the *Annotated Architecture A* that are annotated with the *Security Characteristic* s_C^{CS} of the coupling scenario by at least one *Annotation* α_C

$$\Delta_C^{cs} \neq \emptyset$$

where

$$\Delta_C^{cs} = \{\delta_C \in \Delta_C \mid \exists \delta_A \in \Delta_A \colon \exists \alpha_A^{cs} \in An_A^{cs} \colon \langle \alpha_A^{cs}, \delta_A \rangle \wedge \delta_C \leftrightarrow \delta_A\}$$

IC2.2(T): There must exist *Configurations* cfg_C of the *Annotated Source Code C* which correspond to *Configurations* cfg_A in the *Annotated Architecture A* that use at least one *Annotation* α_C, annotating the *Security Characteristic* s_C^{CS} in the coupling scenario.

$$CFG_C^{cs} \neq \emptyset$$

where

$$CFG_C^{cs} = \{cfg_C \in CFG_C \mid \exists cfg_A \in CFG_A : \exists \alpha_A^{cs} \in An_A^{cs} : \langle cfg_A, \alpha_A^{cs} \rangle \wedge cfg_C \leftrightarrow cfg_A\}$$

In the running example, this IC is fulfilled as *Parameters* of methods in the implementation can be resolved to *Parameters* in services. On the model level, this is shown for the *userData* in the method *store* (*Annotated Source Code*) which can be resolved to the parameter *userData* in the service *store* (*Annotated Architecture*).

Resolving the annotation in the *Annotated Architecture* from the *Annotated Source Code* that annotates the security characteristic in the *coupling scenario* requires the security characteristic, system element and configuration from **IC1** and **IC2**. Counter-examples in the *Source Code Analysis Result* are based on the annotations in the *Annotated Source Code*. Consequently, to enforce the resolution of the annotation in the *Annotated Architecture*, an annotation in the *Annotated Source Code* must exist, following **IC3**.

IC3(C): There must exist an *Annotation* α_C in the *Annotated Source Code C* that annotates a *Security Characteristic* $s_C^{cs} \in S_C^{cs}$ affected by **IC1** to a *System Element* $\delta_C^{cs} \in \Delta_C^{cs}$ affected by **IC2** and is referenced by a *Configuration* $cfg_C^{cs} \in CFG_C^{cs}$ also captured in **IC2**.

$$An_C^{cs} \neq \emptyset$$

where

$$An_C^{cs} = \{\alpha_C \in An_C \mid \exists \delta_C^{cs} \in \Delta_C^{cs}, \exists s_C^{cs} \in S_C^{cs}, \exists cfg_C^{cs} \in CFG_C^{cs} : \langle \alpha_C, \delta_C^{cs} \rangle \wedge \langle \alpha_C, s_C^{cs} \rangle \wedge \langle cfg_C^{cs}, \alpha_C \rangle\}$$

IC3 introduces an implicit correspondence between annotations in the *Annotated Architecture* and annotations in the *Annotated Source Code* based on the assumption that an annotation can be identified by the system element, security characteristics and configuration (see Subsubsection 5.1.2.2). Additionally, although the system elements and security characteristics used in the annotation describe the system, the annotations themselves only provide the means to connect this information. Consequently, formulating **IC3** as *constraint condition* rather than a *correspondence condition* allows the exchange of the annotation mechanisms in an analysis while maintaining the resolution capability. In the running example, **IC3** holds because the *Annotated Source Code* contains an annotation, which annotates the *confidentiality level* such as the value *high* (**IC1**) to *Parameters* such as *userData* (**IC2**) while using an implicit configuration (**IC2**).

Specification-based source code analyses only report vulnerabilities for specific security characteristics if these security characteristics are part of the security policy of the analysis.

Thus, the security policy of the source code analysis must use the security characteristics affected by **IC1** to ensure that the analysis reports annotations for these security characteristics in the *Source Code Analysis Result*. To ensure that these security characteristics (and the corresponding values) are available, the *coupling expert* has to investigate whether a transformation of the security policy of the architectural analysis to the security policy of the source code analysis can be defined. If the *Annotated Architecture* does not comprise enough information for the *coupling expert* to provide the security policy by a transformation, the security policy has to be manually created in the *Completion* transformation. For this purpose, the *coupling expert* and the *analysis architects* of both analyses to be coupled have to define a security policy for the source code analysis which fits to the architectural security policy. We guide the manual creation and the transformation by **IC4**.

IC4(C): *Security Policies* $pol_C \in Pol_C$ of the source code analysis must use the *Security Characteristics* $s_C^{cs} \in S_C^{cs}$ affected by **IC1** of their *Configuration* $cfg_C \in CFG_C$.

$$\forall pol_C \in Pol_C : \forall cfg_C \in CFG_C, \forall s_C^{cs} \in S_C^{cs} : \langle cfg_C, pol_C \rangle \wedge \langle cfg_C, s_C^{cs} \rangle \Rightarrow \langle pol_C, s_C^{cs} \rangle$$

IC4 ensures that the security characteristics affected by **IC1** are part of the security policy of the source code analysis. Consequently, following **IC4** avoids encounters with security characteristics of data later in the coupling process which do not correspond to the *Annotated Architecture*. **IC4** holds in our running example because the security policy captures the security characteristic confidentiality level, and the values *low*, *medium*, and *high*, all of which can be resolved to the *Annotated Architecture*.

Ensuring that all relations between the security characteristics in the security policy of the source code analysis correspond to relations in the *Annotated Architecture* would be beneficial. However, this requirement would be too restrictive because the security policy of the architectural analysis may not capture all relations required in the source code analysis. The running example illustrates this case with the relation between the security policies of the *Annotated Architecture* and the *Annotated Source Code*. The security policy in the *Annotated Source Code* describes the allowed flow relations between the values of the confidentiality levels. In contrast, the security policy in the *Annotated Architecture* describes the relations between the values of the confidentiality levels and the locations of the resources. Thus, we cannot establish correspondences between the relations of both security policies. In these scenarios, the security policy in the *Annotated Source Code* must be created manually. Consequently, this creation must include information from the analysis architect of the architectural analysis and the coupling expert to ensure that the security policy contains security characteristics according to **IC4**.

5.1.3.2. Integration Conditions: Non-Compliance Analysis

In the *Parameterization Model Resolution* transformation, the values of security characteristics resulting from the implementation in the *Parameterization Model* are extracted from

the *Source Code Analysis Result*. Thus, we have to ensure that system elements, the security characteristics, and the configuration in the *Source Code Analysis Result* are resolvable to the elements of the *Annotated Source Code* affected by **IC1** and **IC2** respectively.

In the reference metamodels, we do not specify which security characteristics or values must be part of the counter-example. Consequently, a *Source Code Analysis Result* could only contain security characteristics or system elements that are not resolvable to the *Annotated Architecture*, making it impossible to resolve the annotations in the *Annotated Architecture*. In our running example, this means that a counter-example could only contain flows to *fields*. To avoid the sole availability of system elements and security characteristics that are not resolvable to the *Annotated Architecture*, we specify **IC5**.

IC5(T): There must exist *Security Characteristics* $s_R \in S_R$ and *System Elements* $\delta_R \in \Delta_R$ in the *Source Code Analysis Result R* corresponding to *Security Characteristics* $s_C^{cs} \in S_C^{cs}$ and *System Elements* $\delta_C^{cs} \in \Delta_C^{cs}$ of the *Annotated Source Code C* that are affected by **IC1** and **IC2** respectively.

$$\Delta_R^{cs} \neq \emptyset \wedge S_R^{cs} \neq \emptyset$$

where

$$\Delta_R^{cs} = \{\delta_R \in \Delta_R \mid \exists \delta_C^{cs} \in \Delta_C^{cs} : \delta_R \leftrightarrow \delta_C^{cs}\}$$
$$S_R^{cs} = \{s_R \in S_R \mid \exists s_C^{cs} \in S_C^{cs} : s_R \leftrightarrow s_C^{cs}\}$$

IC5 does not require that the security characteristic and system element to be comprised in the same *Result Entry Element*. Enforcing at least one of such a *Result Entry Element* with **IC6** is necessary; otherwise, only *Result Entry Elements* could exist in which either the system element or the security characteristic can be resolved to the *Annotated Architecture*. Such constellations would make resolving annotations in the *Annotated Source Code* impossible. Note that **IC6** still allows counter-examples for flows from or to *Fields*.

IC6(C): There must exist a *Result Entry Element* $ree \in REE$ in the *Source Code Analysis Result R* containing a *System Element* $\delta_R^{cs} \in \Delta_R^{cs}$ and a *Security Characteristic* $s_R^{cs} \in S_R^{cs}$, both affected by **IC5**.

$$\exists ree \in REE : \exists \delta_R^{cs} \in \Delta_R^{cs}, s_R^{cs} \in S_R^{cs} : \langle ree, \delta_R^{cs} \rangle \wedge \langle ree, s_R^{cs} \rangle$$

IC5 and **IC6** both hold in the running example as the *Result Entry* contains the *Result Entry Element* containing the parameter *userData* of the method *store* and the confidentiality level *low*. Both can be resolved to the correspondingly named elements in the *Annotated Architecture*. In addition to the system element and a security characteristic, we require with **IC7** a configuration resolvable to the *Annotated Architecture* that enables the calculation of the annotations.

IC7(T): There must exist *Configurations* $cfg_R \in CFG_R$ in the *Source Code Analysis Result* R corresponding to *Configurations* $cfg_C^{cs} \in CFG_C^{cs}$ of the *Annotated Source Code* C affected by **IC2**.

$$CFG_R^{cs} \neq \emptyset$$
where
$$CFG_R^{cs} = \{cfg_R \in CFG_R \mid \exists cfg_C^{cs} \in CFG_C^{cs} \colon cfg_R \leftrightarrow cfg_C^{cs}\}$$

IC7 holds in the running example as both analyses use *implicit* configurations, allowing to determine their correspondence by the applied models used in the coupling.

5.1.3.3. Integration Conditions: Parameterization

All elements in the *Parameterization Entries* of the *Parameterization Model* must be resolvable to elements of the *Annotated Architecture* for achieving the coupling. For calculating the annotation to be modified, we must resolve the system element and configuration in the *Annotated Architecture*. **IC5** and **IC6** do not impose any restriction that the *Source Code Analysis Result* must contain only security characteristics and system elements resolvable to the *Annotated Architecture*. Consequently, a *Parameterization Model* could be constructed that only contains fields. Such a *Parameterization Model* would not be applicable in the *Parameterization* step in our running example as fields are not represented in the component-based *Annotated Architecture*. Furthermore, the configuration used in the *Parameterization Model* must be resolvable to the *Annotated Architecture* to ensure that the annotations can be calculated for the correct system. Thus, **IC8** ensures that the *Parameterization Model* provides the capability for all configurations and system elements by utilizing the elements in the *Source Code Analysis Result* corresponding to **IC2**.

IC8(C): All *Parameterization Entries* $pe \in PE$ in the *Parameterization Model* PM have to contain *System Elements* $\delta_{PM} \in \Delta_{PM}$ and *Configurations* $cfg_{PM} \in CFG_{PM}$ which must correspond to *System Elements* $\delta_R^{cs} \in \Delta_R^{cs}$ and *Configurations* $cfg_R^{cs} \in CFG_R^{cs}$ in the *Source Code Analysis Result* R, which are affected by **IC5** and **IC7**, respectively.

$$\forall pe \in PE \colon \exists \delta_{PM} \in \Delta_{PM}, \exists cfg_{PM} \in CFG_{PM}, \exists \delta_R^{cs} \in \Delta_R^{cs}, \exists cfg_R^{cs} \in CFG_R^{cs} \colon$$
$$\langle pe, \delta_{PM} \rangle \wedge \langle pe, cfg_{PM} \rangle \wedge cfg_{PM} \leftrightarrow cfg_R^{cs} \wedge \delta_{PM} \leftrightarrow \delta_R^{cs}$$

IC8 can be established in our running example by providing an *Parameterization Model* which only contains the system element *Parameter*, as these are the only annotated elements in the *Annotated Architecture* that are affected by **IC2**. **IC8** can also be established in our running example if the *Parameterization Model* provides the implicit configuration, e.g., by adopting it from the *Source Code Analysis Result*.

IC5 and **IC6** require at least one *Result Entry* to provide a security characteristic and system element which are resolvable to the *Annotated Architecture*. Thus, a *Parameterization*

Model and *Parameterization Model Resolution* could be constructed so that the *Parameterization Model* contains security characteristics that cannot be resolved to the *Annotated Architecture* (e.g., locations in the prior example). In addition, when the interpretation in the *Parameterization Model Resolution* calculates values unaffected by **IC1**, a corresponding element in the *Annotated Architecture* cannot be resolved. In our running example, a *analysis architect* unaware of the semantics of the security characteristics used in the *Annotated Architecture* could create a *Parameterization Model Resolution* that calculates the value *Low;Medium;High* instead of *high* for the parameter *userData*. However, this value cannot be mapped to the *Annotated Architecture* as it only contains the values *high*, *medium*, and *low*.

The information which security characteristic is resolvable to the security characteristic in the coupling scenario, and the information which values are allowed for it is available in the *Annotated Source Code* due to **IC1**. Consequently, we formulate **IC9** to ensure that the security characteristics and their calculated values captured in the *Parameterization Model* can be resolved to the *Annotated Architecture*.

IC9(T): All *Parameterization Entries* $pe \in PE$ in the *Parameterization Model PM* have to contain *Security Characteristics* $s_{PM} \in S_{PM}$ corresponding to *Security Characteristics* $s_C^{cs} \in S_C^{cs}$ in the *Annotated Source Code* affected by **IC1**.

$$\forall pe \in PE\colon \exists s_{PM} \in S_{PM}, \exists s_C^{cs} \in S_C^{cs}\colon \langle pe, s_{PM}\rangle \wedge s_{PM} \leftrightarrow s_C^{cs}$$

Because **IC8** and **IC9** are defined for all elements in the *Parameterization Model*, all prior input and output metamodels must contain corresponding elements.

With **IC9**, a configuration in the *Annotated Architecture* may not support the values of security characteristics in the *Parameterization Model*. It may be possible that the value *medium* in the prior example is defined in the *Annotated Architecture*, but the configuration resolved from the *Parameterization Model* does not consider it for analysis. Thus, we constrain the security characteristics in an *Parameterization Model* with **IC10** to ensure that the values of the security characteristics are supported by the configuration in the *Annotated Architecture*.

IC10(C): The *Security Characteristics* $s_C^{cs} \in S_C^{cs}$ in the *Annotated Source Code C* corresponding to the *Security Characteristic* $s_{PM} \in S_{PM}$ in a *Parameterization Entry* $pe \in PE$ in the *Parameterization Model PM* affected by **IC9** must be referenced in a *Configuration* $cfg_C^{cs} \in CFG_C^{cs}$ in the *Annotated Source Code C* corresponding to the *Configuration* $cfg_{PM} \in CFG_{PM}$ affected by **IC8** in the same *Parameterization Entry*.

$$\forall pe \in PE\colon \exists s_{PM} \in S_{PM}, \exists cfg_{PM} \in CFG_{PM}, \exists s_C^{cs} \in S_C^{cs}, \exists cfg_C^{cs} \in CFG_C^{cs}\colon$$
$$\langle pe, s_{PM}\rangle \wedge \langle pe, cfg_{PM}\rangle \wedge cfg_{PM} \leftrightarrow cfg_C^{cs} \wedge s_{PM} \leftrightarrow s_C^{cs} \wedge \langle cfg_C^{cs}, s_C^{cs}\rangle$$

IC10 would hold in our running example, if the *Parameterization Model* only includes the values *high*, *medium* and *low* from the security characteristic confidentiality level. This IC

holds in this case because the implicit configuration in the *Annotated Source Code* contains corresponding values.

5.1.4. Specializing the Coupling Process for Specification-Based Analysis Coupling

We now detail the specialization of the coupling process (introduced in our coupling design in Section 4.4) for the coupling of architectural analyses with *specification-based* source code information flow analyses. The transformations in our process are assumed to capture correspondences which reflect the ICs; thus establishing resolution capabilities between the corresponding elements. We present the specialization of the coupling process for each process step in the following.

5.1.4.1. Process Step: Analyses Input Model Consistency

The *Consistency* transformation and *Completion* transformation of the *Analyses Input Model Consistency* step establish consistency between the system elements in the *Annotated Architecture* and the *Source Code*, and complete the source code to the final system respectively (see Subsubsection 4.4.3.1). However, we extend the responsibilities of these transformations to address the specifications in the *Annotated Architecture* and *Annotated Source Code*. In particular, the *Consistency* transformation is critical because it enables that information flows reported in the *Source Code Analysis Result* indicate non-compliance between the architectural specification and the implementation.

Consistency Transformation We extend the *Consistency* transformation, so that information flows reported in the *Source Code Analysis Result* indicate non-compliance regarding the specification in the *Annotated Architecture*. This extension is based on the property of the source code analysis that counter-examples reported in the *Source Code Analysis Result* indicate non-compliance between the implementation and the specification in the *Annotated Source Code*. We exploit this property for our coupling by establishing consistency between the *Annotated Architecture* and the *Annotated Source Code*, so that they represent the same system and specification. If this consistency exists, a counter-example in the *Source Code Analysis Result* also indicates non-compliance between the values specified in the *Annotated Architecture* and those resulting from the implementation.

This approach is exemplified by our running example (see Chapter 3). In the running example, the *Annotated Architecture* and the *Annotated Source Code* represent the same system (illustrated in partial model (a) in Figure 3.1 and partial model (d) in Figure 3.2). In this example, consistency is shown by the same names and corresponding references of system elements in both models. For example, the component *OnlineShop* in Figure 3.1 is consistent with the correspondingly named class *OnlineShop*. The service *buy* in the *OnlineShop* component is consistent with the method *buy* in the corresponding class. This consistency also results in the parameters to be consistent, e.g., the parameter *userData*

in the service *buy* is consistent with the parameter *userData* of the method *buy* in the *Annotated Source Code*. Our running example also illustrates consistency between the annotated values of the security characteristics of data. For example, the parameter *userData* of the service *buy* and the parameter *userData* of the method *buy* are both annotated with the value *high*. The information flows reported by the source code analysis indicate that the parameter *userData* in the method *store* now contains data classified as *high* (as discussed in Subsection 5.1.1). This *value resulting from the implementation* does not comply with the value *low* specified for this parameter in the *Annotated Source Code*. Thus, the information flow reported by the source code analysis results in non-compliance between the implementation and the specification in the *Annotated Source Code*. Due to the consistency between the parameters *userData* in the method *store* in the *Annotated Source Code* and the parameter *userData* in the service *store* in the *Annotated Architecture*, we can interpret them as representing the *same* system element (semantically identical). Consequently, the information flow reported in the *Source Code Analysis Result* also results in the parameter *userData* in the service *store* to contain *high* data. This value also does not comply with the value *low* specified for this parameter in the *Annotated Architecture*.

To enable this interpretation, annotations in the *Annotated Source Code* must exist that correspond to the annotations in the *Annotated Architecture*. This necessary relation is described by **IC3**. The annotations must annotate the same values of the security characteristic of the coupling scenario to corresponding system elements in the *Annotated Source Code* and the *Annotated Architecture*. Furthermore, the annotations must be included in configurations that can be related to each other. For achieving **IC3**, the *Consistency* transformation must first establish relations prescribed by **IC1** and **IC2**. First, consistency must be established between the values of the security characteristic of the coupling scenario in the *Annotated Architecture* and values of a corresponding security characteristic in the *Annotated Source Code* (**IC1**). In the running example, the transformation establishes consistency between the values *high*, *medium* (not illustrated in partial model (a) of Figure 3.1 for brevity), and *low* in the *Annotated Architecture*, and corresponding values in the *Annotated Source Code*. Then, consistency has to be established between system elements in the *Annotated Source Code* and system elements in the *Annotated Architecture* that are annotated with the security characteristic of the coupling scenario (**IC2**). This consistency is necessary because non-compliance can only be detected for annotations that annotate corresponding system elements in the *Annotated Architecture* and the *Annotated Source Code* (see Definition 1). In our running example, the *Consistency* transformation has to provide consistency between the parameters of the services in the *Annotated Architecture* that are annotated with the *Confidentiality Levels* and the parameters of the corresponding methods in the *Annotated Source Code*.

Non-compliance is calculated for a single *Configuration* in the *Annotated Architecture* as it defines the system under analysis and the applied specification. Thus, the *Consistency* transformation has to establish consistency for the configurations used in the architectural analysis and the configurations in the source code analysis (**IC2**). The *Source Code Analysis Result* identifies the configuration in the *Annotated Source Code* used to calculate the information flows which do not comply with the source code specification. Thus, the consistency between the configurations of the *Annotated Architecture* and the *Annotated*

Source Code also establishes a relation between the *Annotated Architecture* and the *Source Code Analysis Result*. This transformation is influenced by whether the analyses use *explicit* or *implicit* configurations (described in paragraph 5.1.2.1). If both analyses use *explicit* configurations, the transformation has to ensure that the configuration in the *Annotated Source Code* includes the values of security characteristics and the system elements corresponding to the elements of the configuration in the *Annotated Architecture*. As soon as either analysis uses an *implicit* configuration, the coupling expert has to provide the capability to relate the configurations by capturing all models used for the *implicit* configuration (**IC2**). This capability can be achieved, for instance, by a coupling tool that captures the models used by either analysis for an execution of the coupling process. We provide a publicly available prototypical coupling tool [268] with this capability.

With the consistency established for the system elements, values of security characteristics and configurations, consistency can be established between the annotations in the *Annotated Architecture* and the *Annotated Source Code*. For every system element in the *Annotated Architecture* annotated with the value of the security characteristic in the coupling scenario, an annotation in the *Annotated Source Code* must exist annotating the corresponding system element with a corresponding value. The affected annotations in the *Annotated Source Code* must be included in the configurations of the source code analysis corresponding to the configuration in the *Annotated Architecture*. Through this approach, annotations exist in the *Annotated Source Code* that (1) annotate a security characteristic corresponding to the one of the coupling scenario in the *Annotated Architecture*, (2) to a system element corresponding to those in the *Annotated Architecture* annotated with the security characteristic of the coupling scenario, and (3) which are supplied to configurations that can be related to the correct configuration in the *Annotated Architecture* (**IC3**).

IC2 and **IC3** do not require to establish consistency *for all* values of the security characteristic in the *Annotated Architecture* to detect non-compliance. However, calculating the values resulting from the implementation in the *Parameterization Model Resolution* transformation requires selecting new values as we will discuss in the *Parameterization* step later in this section. These values must be resolvable to values in the *Annotated Architecture*. Thus, we propose the *Consistency* transformation to establish consistency between all values of the security characteristic of the coupling scenario in the *Annotated Architecture* and values of the corresponding security characteristic in the *Annotated Source Code*. This consistency avoids limiting the capability of the *Parameterization Model Resolution* transformation to select new values in the *Annotated Source Code*.

Note that we rely in this coupling approach on the capability that this consistency can be established automatically. Listing 5.1 presents a pseudocode to illustrate the transformations for our running example addressing **IC1**, **IC2** and **IC3**. This example code results, amongst others, in the following transformations regarding elements in the running example:

1. Mapping the implicit configuration $config_A$ of the architectural analysis to a configuration $config_C$ of the source code analysis (**IC2**).

```
procedure Consistency(AAM, config_A)
IC2 config_C ← map config_A to configuration of
         source code analysis
    Values_A ← get values of security characteristic regarding coupling scenario from config_A
IC1 Levels_C ← map values of Values_A to
         security levels of source code analysis

    ∀component ∈ Components of AAM:
      map component to class class_C ∈ Source Code

      ∀providedService ∈ Provided Services of component:
        map providedService to method method_C ∈ Methods of class_C

        ∀parameter_A ∈ Parameters of method_C:
IC2       map parameter_A to parameter
                    parameter_C ∈ Parameters of method_C

          {annotation_A ∈ Annotations of config_A| annotation_A is information flow annotation ∧
              annotation_A annotates parameter_A} ≠ ∅ ⇒
            get value_A ∈ Values_A annotated to parameter_A
            get level_C ∈ Levels_C corresponding to value_A
IC3         generate annotation_C ∈ Annotations of
              config_C annotating level_C to parameter_C
end procedure
```

Listing 5.1: Example pseudo code for the Consistency transformation of the running example using the Annotated Architecture *AAM* and the configuration $config_A$ of the architectural analysis. Sets of elements start with an upper-case while single elements start with a lower-case. Subscripts A and C mark elements of architecture and code respectively. Lines marked with IC*i* establish relations to fulfill integration condition *i*.

2. Mapping the *confidentiality levels high*, *medium* (not shown in Figure 3.1 for the architecture), and *low* contained in $Values_A$ to corresponding values available to the source code analysis, captured in $Levels_C$ (**IC1**).
3. Mapping the *component OnlineShop* to a *class OnlineShop* (**IC2**).
4. Mapping the provided service *buy* (*providedService*) of *OnlineShop* with the parameters *userData* and *cart* ($parameter_A$) to a *method buy* in the class *OnlineShop* with the parameters *userData* and *cart* ($parameter_C$) and an empty body (**IC2**).
5. Generating annotations $annotation_C$, annotating, for example, the parameter *userData* ($parameter_C$) corresponding to the parameter *userData* ($parameter_A$) with the value *high* ($level_C$) corresponding to the annotated value high ($value_A$) in the architecture (**IC3**).

Consistency for security policies in the *Annotated Architecture* and the *Annotated Source Code* can be established if they capture *at least* the same relations between the values of the security characteristic of the coupling scenario. In this case, the *Consistency* transformation can automatically ensure that the values of the security characteristic in the coupling scenario are used in the security policy of the source code analysis (**IC4**). This property *does not hold* in our running example because no relations between the *confidentiality*

levels are provided as part of the security policy. Consistency can also be established if the architectural analysis uses an *integrated* security policy as long as the prior described property holds. Here, the coupling expert has to interact with the analysis architect of the architectural analysis to determine the relations realized in the integrated security policy. For example, the analysis of Gerking and Schubert [96] is based on the premise that confidential information must not flow to non-confidential locations. With this knowledge, the coupling expert can generate a security policy used by the source code analysis in the running example, which captures the relation that data in system elements classified as *high* must not flow to system elements classified as *low*.

The *Consistency* transformation can provide a *correspondence model* to capture all correspondences between the elements in the *Annotated Architecture* and the *Annotated Source Code* established in the transformation. This correspondence model can be used in the coupling to resolve elements in the *Annotated Architecture* to represent the values of the security characteristic in the coupling scenario resulting from the implementation. The *correspondence metamodel* captures the semantically identical relation and has to be created by the *coupling expert* specific for the coupling scenario (i.e., for the *metamodels* of both analyses). Correspondence models allow reusing the correspondences in later evolution steps and avoid the need for model traversal to resolve correspondences based on heuristics. However, a *coupling expert* can choose to omit the *correspondence models* if simple heuristics can be defined to enable the resolution. For example, a heuristic may resolve a component in the *Annotated Architecture* based on the names of classes in the *Annotated Source Code* and vice versa if every component corresponds to a class with the same name (as exemplified by [173, 181]).

Completion Transformation The *Completion* transformation must complete the implementation, the specification and the configurations after the *Consistency* transformation to sufficiently analyze the implementation for information flows resulting in non-compliance. In particular, omitting completion of specifications for implementation elements without corresponding architectural elements can lead to missed detection of non-compliance or false positives in the *Source Code Analysis Result*.

Implementation elements may not have corresponding architectural elements in the *Annotated Architecture* (e.g., *fields* in component-based architectural models) or are abstracted (e.g., internal procedures may retrieve sensitive data from data sources such as databases or files for internal calculations). These elements are also an example for the reason why we cannot establish consistency for all implementation elements in the *Annotated Source Code* during the *Consistency* transformation. These elements are likely to be completed in the *Completion* transformation to provide a working system.

Obviously, no consistency for the annotations of the security characteristics of data for these implementation elements to the *Annotated Architecture* can be established. However, non-compliance can arise from information flows originating from implementation elements without corresponding architectural element. Thus, without completing these annotations, the source code analysis cannot detect these information flows. For example,

the implementation in our running example realizes an information flow from the field *privilegeSum* to the parameter *userData* in the method *store* (shown in partial model (d) of Figure 3.2). This information flow results in non-compliance because data in *privilegeSum* classified with the value *medium* is propagated to the parameter *userData* of the method *store* expected to contain data classified with the value *low*. Consequently, using an implementation only with *system elements* for which correspondences exist would result in the source code analysis to miss this information flow.

The analysis may also provide false positive results if the specifications are not completed. The method *encrypt* in the *DataStorage* of our running example would not be considered a declassification by *JOANA* if not specified as such. Without specifying this declassification, *JOANA* would report an information flow from the *high* classified parameter *userData* in the method *store* of the class *DataStorage* to the *low* classified parameter *log* in the method *log*, propagated by the method *store* in the class *DataStorage*. However, the encryption in the method *store* of the *DataStorage* sufficiently declassifies data from *high* to *low* by using the cryptographic algorithm *AES*. Consequently, a false positive information flow would be reported.

The *Completion* transformation must also provide the security policy of the source code analysis if consistency cannot be established in the *Consistency* transformation. In this case, the coupling expert must ensure that the security policy in the *Annotated Source Code* provides relations for the values of the security characteristic corresponding to the values of the coupling scenario in the *Annotated Architecture* (**IC4**). Without ensuring this capability, the source code analysis cannot analyze the implementation for information flows resulting in non-compliance.

As a result, it is essential for the *Completion* transformation to provide (1) implementation elements without correspondence to the *Annotated Architecture*, *and* (2) annotations for these implementation elements specifying values of security characteristics which are consistent with the values of security characteristics of the coupling scenario in the *Annotated Architecture*, and (3) the security policy that captures values corresponding to the values of the security characteristic of the coupling scenario.

5.1.4.2. Non-Compliance Analysis

In the *Non-Compliance Analysis* step, the source code analysis is executed to inspect the *Annotated Source Code* for information flows that violate the given specification based on the provided security policy. These flows are reported as counter-examples in the *Source Code Analysis Result*. In the running example, two information flows to the parameter *userData* specified with the value *low* in the method *store* are reported (partial model (f) of Figure 3.2): One information flow from the field *privilegeSum* with the confidentiality level *medium* and one from the parameter *userData* with the confidentiality level *high* in the method *buy*. This is because both information flows violate the security policy (partial model (e) of Figure 3.2) that only data with lower confidentiality must flow to data with higher confidentiality. Another information flow is reported from the parameter *userData*

with the confidentiality level *high* in the method *buy* to the parameter *entry* with the confidentiality level *low* in the method *log* due to the same reason. Each counter-example results in non-compliance between the specifications in the Annotated Source Code and the implementation because the sink implementation element contains data classified with another value of the security characteristics of data than specified. In our running example, *store* contains data of the values *medium* and *high* in contrast to the data with the expected values *low*. Due to the *Consistency* transformation in the *Analyses Input Model Consistency* step, every information flow reported in the *Source Code Analysis Result* for a sink implementation element that corresponds to an architectural element annotated with the security characteristic of the coupling scenario results in non-compliance with the architectural specification.

Source code analyses commonly do not provide correspondence models between the elements of the *Source Code Analysis Result* and the *Annotated Source Code*. However, the elements in counter-examples refer to elements in the *Annotated Source Code*, e.g., by the fully qualified name of a class, the method name and the parameter name to identify a parameter. The coupling expert can exploit this information to resolve the elements in the *Source Code Analysis Result* to the corresponding elements in the *Annotated Source Code*. We illustrate this capability in the evaluation of our coupling approach (see Chapter 8) by reconstructing correspondence models between the *Source Code Analysis Result* and the *Annotated Source Code* based on the information provided in the output of the source code analyses.

The model elements in the *Source Code Analysis Result* and their relations to the *Annotated Source Code* must fulfill the ICs defined for the coupling scenario. However, because the black-box view of the source code analysis does not allow enforcing the ICs, the coupling expert must ensure that the source code analysis is suitable for the coupling scenario by interacting with the analysis architect.

5.1.4.3. Parameterization

The *Parameterization* step is specific to the non-compliance definition presented in Subsection 5.1.1. Thus, we describe the capabilities of the *Parameterization Model Resolution* transformation and the *Integration* transformation specific to the specification-based coupling approach in the following.

Parameterization Model Resolution The *Parameterization* step requires obtaining the values resulting from the implementation. In the specification-based coupling approach, the values resulting from the implementation can already be calculated in the *Parameterization Model Resolution* transformation. Calculating the values already in the *Parameterization Model Resolution* is possible because all information required is available on the source code level because of the prior process steps (*Analyses Input Model Consistency*, *Non-Compliance Analysis*) which exploit the semantically identical relation type. We discuss this capability in the following.

The *Analyses Input Model Consistency* step established consistency between the specifications of the *Annotated Architecture* and the *Annotated Source Code* so that they represent the same specification for the system under analysis. Consequently, all information flows reported in the *Source Code Analysis Result* involving an implementation element that corresponds to an architectural element are also non-compliant information flows between the implementation and the specification in the *Annotated Architecture*. Furthermore, due to the semantically identical relation type and the *Analyses Input Model Consistency* step, all information to calculate the *values resulting from the implementation* are already available on the source code level (i.e., in the *Source Code Analysis Result* and the *Annotated Source Code*). This information includes the security policy, the available values of the security characteristics of data in the *Annotated Source Code*, and the detected information flows in the *Source Code Analysis Result*. Due to the consistency established in the *Analyses Input Model Consistency* step, it is not necessary to bridge the gap between the architectural level and the source code level to determine non-compliance and to calculate the values resulting from the implementation.

We propose the following approach for calculating the values resulting from the implementation for a single information flow: If a single information flow to a sink implementation element is reported, the value of the security characteristic of data resulting from the implementation is the value specified to the source. This is possible due to the interpretation of the reported information flow described in Subsection 5.1.1. In our running example, this approach results in the value *high* for the parameter *entry* in the method *log* due to the information flow from the parameter *userData* in the method *buy*. An information flow that originates from a source implementation element which corresponds to an architectural element results from the implementation in a value that conforms to **IC9**. This is because we also establish consistency between the annotated values of the security characteristics of data in the *Annotated Architecture* and the *Annotated Source Code* in the *Consistency* transformation for such annotations.

However, a *Source Code Analysis Result* may contain several counter-examples with the same implementation element as sink. In our running example, this is exemplified by the information flows from the parameter *userData* in the method *buy* and the field *privilegeSum* to the parameter *userData* in the method *store* (see partial model (d) of Figure 3.2). These information flows are reported as two counter-examples in the *Source Code Analysis Result* (see partial model (f) in Figure 3.2). Consequently, the calculation of the values resulting from the implementation has to consider multiple information flows to determine a value representing both flows. For strictly ordered *lattices* (used in our running example; see partial model (e) of Figure 3.2) we propose to select the most security critical value assigned to the source implementation elements of the information flows reported in the *Source Code Analysis Result* for a single sink implementation element as the value resulting from the implementation. For this purpose, the *Parameterization Model Resolution* first collects all values of the security characteristics of data of the coupling scenario from the sources of the flows in the *Source Code Analysis Result*. Then, the *Parameterization Model Resolution* selects the value resulting from the implementation by following the lattice. The value is selected to which no allowed flow from the other values exist (i.e., the most security critical value).

This approach is based on the following interpretation: A strictly ordered lattice for confidentiality policies implies that data which is allowed to flow to other data has either the same confidentiality level or a lower confidentiality level assigned. Thus, an actor in the system that is allowed to access data with a certain value may access all data with values to which flows are allowed to. For example, an actor that is allowed to access data of the value *high* (highly confidential) is also allowed to access data with the value *medium* and *low* (because *high* is the highest value in the lattice). However, if the actor accesses data with the value *high*, security is violated because no flow is allowed from *high* to *medium*. With this interpretation, an actor that is allowed to access data assigned with this value is also allowed to access data assigned with all other values of the sources of the flows. In our running example, this approach results in the level *high* for the parameter *userData* in the *store* service. If a security issue arose due to value *medium*, it also arises because of *high*.

For other semantics of values of security characteristics of data, the calculation of the values resulting from the implementation requires different interpretations or application of heuristics as shown by our preliminary research in the master's thesis of Lehmann [185]. More comprehensive research and constructive guidelines are subject to future work (see Section 11.3).

Using multiple counter-examples may require calculating values of security characteristic of data not contained in the *Source Code Analysis Result* because it only contains values used in the counter-examples. Note that this calculation is not necessary for our running example because the *Source Code Analysis Result* already contains all values available in the *Annotated Source Code* (*low*, *medium* and *high*). We address this challenge by using the *Annotated Source Code* as additional input for the *Parameterization Model Resolution* transformation to verify if a calculated new value exists in the configuration used for the source code analysis. The coupling expert has to verify that the calculated values resulting from the implementation correspond to values available in the *Annotated Architecture* (**IC9**).

The *Parameterization Model Resolution* transformation must also provide the configuration applied in calculating the counter-example. This configuration must correspond to configurations in the *Annotated Architecture* (**IC8**). This correspondence ensures that the annotations and security characteristics used in the source code analysis correspond to those in the architectural analysis. The *Completion* transformation, for instance, could create new values of the security characteristics of data in the *Annotated Source Code*. The coupling expert has to ensure that the *Parameterization Model Resolution* transformation verifies the existence of values in the respective configuration in the *Annotated Architecture* which correspond to the security characteristic in every *Parameterization Entry* (**IC10**).

Integration Based on the *Parameterization Model*, the *coupling expert* can define the *Integration* transformation. We designed the *Parameterization Model* so that every *Parameterization Entry* contains only the information used to resolve the annotation to be modified. Thus, the *coupling expert* describes the transformation so that it searches for

every annotation in the *Annotated Architecture* which fulfills the following three criteria: (1) the annotation annotates a value of the type of the security characteristic of data in the coupling scenario corresponding to the type of security characteristic of data captured in the *Parameterization Entry*, (2) the annotation annotates the architectural element which corresponds to the implementation element provided in the *Parameterization Model* and (3) the annotation is defined in the configuration in the *Annotated Architecture* which corresponds to the configuration provided in the *Parameterization Model*.

If the prior transformations conform to our ICs the relations between the *Parameterization Model* and the *Annotated Architecture* for these three criteria are ensured. In our running example, this approach results in obtaining the annotation for the *userData* parameter in the *store* method of the *DataStorage* component and the annotation for the *entry* parameter in the *log* method of the *Logger* component. Because every information flow calculated by the source code analysis indicates non-compliance with the specifications in the *Annotated Architecture*, no check for non-compliance has to be performed in the *Integration* transformation. Also, the *Parameterization Model* already contains the values resulting from the implementation. Consequently, the *Integration* transformation has to perform two tasks when the respective annotations are resolved. First, the *Integration* transformation has to resolve the corresponding value of the security characteristic of data. For this value, its existence in the configuration in the *Annotated Architecture* corresponding to the configuration in the *Parameterization Model* has to be verified. If this value does not exist in the respective configuration in the *Annotated Architecture*, the *Parameterization Model Resolution* provided erroneous results. Secondly, the value of the security characteristic of data in the annotations is replaced with the value of the security characteristic of data in the *Annotated Architecture* resolved from the *Parameterization Model*. In our running example, this results in replacing the value *low* annotated to the *userData* parameter in the *store* method and the *entry* parameter in the *log* method with the value *high*.

5.1.5. Discussion: Efforts in the Specification-Based Coupling

We now discuss efforts that arise when realizing the specification-based coupling process. To this end, we distinguish the efforts into *one-time efforts* to create automated transformations and *manual efforts* that must be repeated in every execution of the process.

5.1.5.1. Efforts in the Analyses Input Model Consistency Step

The efforts in the *Analyses Input Model Consistency* step include the definition of the *Annotated Architecture*, the *Consistency* transformation, and the *Completion* transformation.

The only model we assume to be created fully manually in our specification-based coupling process is the *Annotated Architecture*, because we use it as prescriptive model.

The *Consistency* transformation in the *Analyses Input Model Consistency* step is expected to be fully automated because all manual effort is deliberately located in the *Completion* transformation. Establishing consistency automatically can be achieved using consistency preservation approaches (e.g., Vitruvius [167]) or generative approaches (e.g., [24, 120]). A significant effort in the coupling results from the definition of the annotations (i.e., the specification of the security characteristics). In Subsection 5.1.2, we discuss that annotations can be resolved by a security characteristic, the system element and a configuration. **IC1** to **IC3** require correspondences for these elements and the existence of corresponding annotations between the *Annotated Source Code* and the *Annotated Architecture*. Thus, these ICs enable automated transformations that provide annotations in the *Annotated Source Code* from annotations in the *Annotated Architecture*, which annotate the security characteristic of the coupling scenario. We demonstrate automatic generation for elements captured by **IC1** to **IC3** in previous work [273] and illustrated their automatic generation with the pseudocode for the *Consistency* transformation (see Listing 5.1). Furthermore, we show this automatization in our replication package [270] for the evaluation of our coupling approaches in this thesis (see Chapter 8). Consequently, this results in one-time effort to create the *Consistency* transformation for all source code elements corresponding to the *Annotated Architecture*. Additional effort in creating the *Consistency* transformation can arise if it has to be parameterized based on different semantics on the model level because of *model-level semantic refinement*. Although this parameterization of the transformation results in additional transformation rules, they have to be created once for the *Consistency* transformation for a pair of analyses. Examples of such additional rules are handling semantics of *disjunctive roles* and confidentiality levels; exemplified in our replication package [270] for the evaluation of our coupling approaches in this dissertation (see Chapter 8).

The *Completion* transformation in the *Analyses Input Model Consistency* step is expected to be performed fully manually as its purpose is to complete the *Annotated Source Code* for all elements without correspondence to the *Annotated Architecture*. One of the major efforts in the *Completion* transformation is the definition of annotations, not generated in the *Consistency* transformation. In our running example, this is shown for the field *privilegeSum* and its corresponding annotations because there is no corresponding architectural element in the *Annotated Architecture*. Such annotations have to be performed manually in every application of the coupling approach. Polikarpova et al. [261] discuss that software engineers can write simple specifications competently when expressed in a similar syntax. In the context of our specification-based coupling approach, the specifications comprise the annotations of security characteristics of data to system elements, which we assume to be simple. However, providing more exact estimations of effort is challenging because the capability of performing the specification depends on familiarity with the syntax which we cannot determine due to our aim to support arbitrary analyses.

In evolution scenarios, only those parts of the implementation must be provided manually in the *Completion* transformation which do not yet exist or do not fit the new architecture after the *Consistency* transformation. To reduce efforts, it is essential to use approaches that allow the reuse of the existing system as much as possible. For example, using a generative approach for the *Consistency* transformation requires transferring the existing

implementation to the generated one. Performing this transfer manually imposes a significant effort, potentially rendering the coupling unfeasible. This challenge can be addressed by using consistency preservation approaches or by using tools to merge generated and existing source code (e.g., our prototypical implementation [315]). Such approaches allow reductions of the manual effort in an evolution scenario. This reduction is especially important in our coupling process for the manually provided annotations as it avoids losing already performed specifications.

We assume that the *Consistency* transformation only generates model elements for which explicit correspondences to the *Annotated Architecture* can be defined (captured in the *Incomplete Annotated Source Code*). However, our assumption regarding the effort to complete the *Incomplete Annotated Source Code* in the *Completion* transformation is conservative, because the *Consistency* step may comprise automated transformations which create elements *without* explicit correspondence. As a result, the effort to be performed in the *Completion* transformation may be reduced by such automated transformations. For example, the implicit policy of the information flow analysis of Gerking and Schubert [96] can be derived from the description of the analysis and be used to automatically generate a corresponding security policy in the *Annotated Source Code*. We provide examples for automated generation of security policies in the replication package [270] for our evaluation of the coupling approaches in this thesis (see Chapter 8). These examples include generation of lattices with the values *high* and *low* and lattices with *disjunctive roles*.

5.1.5.2. Efforts in the Non-Compliance Analysis Step

The effort in the *Source Code Analysis* step depends on the source code analysis applied. Fully automated analyses like *JOANA* [98] require merely the effort to execute the analysis. We see this as a one-time effort, because the coupling expert executing the coupling must understand the user interface only once. This effort can be even more reduced if a coupling framework is used to execute the source code analysis automatically. This results in a one-time effort to create the program for automated analysis execution. We show the feasibility of this approach by applying our prototypical coupling framework [268] with automated execution of source code analyses in our evaluation (see Chapter 8). Analyses with manual intervention (e.g., the information flow analysis of Greiner et al. [105]) impose repeated efforts because they must be executed manually in each execution of the coupling process.

5.1.5.3. Efforts in the Parameterization Step

The efforts in the *Parameterization* step include the definition of the *Parameterization Model Resolution* transformation and the *Integration* transformation.

The *Integration* and *Parameterization Model Resolution* transformations are designed to be fully automated. The *Parameterization Model Resolution* imposes one-time efforts for the coupling expert and the analysis architect to define the transformation rules between the

Source Code Analysis Result and the *Parameterization Model*. The effort for the *Parameterization Model Resolution* imposes additional one-time efforts for the coupling expert and the analysis architect if different semantics have to be supported. The *Integration* transformation imposes one-time efforts for the coupling expert to define how the annotations in the *Annotated Architecture* are calculated from the *Parameterization Model* and how the values of the security characteristics in these annotations are modified.

In summary, only the creation of the *Annotated Architecture* and the *Completion* transformation always introduce guaranteed manual efforts that must be performed in each execution of the specification-based coupling process. Efforts invested in the *Source Code Analysis* transformation depend on the degree of automated execution of the selected analysis. Every other transformation is expected to be fully automated, resulting in a one-time effort for their creation even when supporting different semantics. Still, the effort for this one-time creation depends on the *experience* of the coupling expert and their *familiarity* with the analyses to be coupled.

5.2. Pattern-search-based Coupling

Here, we describe our coupling approach for coupling architectural analyses with *pattern-search-based* source code analyses to detect architectural vulnerabilities arising from violations of data encryption by the implementation. In this coupling approach, we focus on security characteristics specifying data encryption and on source code analyses detecting patterns that indicate vulnerabilities which invalidate data encryption.

The focus on data encryption and encryption-related vulnerabilities is reasonable, as it is a major topic in security research and industry. Cryptographic failures are listed number four in the OWASP Top 10 list of security risks [242] in 2025. 19 of 120 security DSLs investigated by Krausz et al. [174] use model elements addressing cryptography. Several standards and regulations require *secure* data encryption to protect sensitive data (e.g., [21, 87, 140]). Furthermore, various architectural analyses use specifications of data encryption (e.g., [12, 154, 172, 300, 338]). Equally, many source code analyses exist that investigate vulnerabilities affecting data encryption in the implementation. Zhang et al. [366] list 20 analysis tools that detect misuse of cryptographic APIs. This list can be complemented with commercial analyses such as Snyk [310], CodeQL [100, 214], or SonarQube [313].

With our pattern-search-based coupling approach, we illustrate how analyses can be coupled that contain *semantically related* information in the *Annotated Architecture* and the *Source Code Analysis Result*. We select analyses and the security characteristics such that elements for representing the system are semantically related and elements for representing security-related information are semantically related. The security characteristics in the *Annotated Architecture* describe the need for data encryption while the *Source Code Analysis Result* contains patterns that indicate vulnerabilities invalidating data encryption. In addition, we focus on annotations annotating the security characteristic of the coupling scenario to architectural elements that represent communication (e.g., linking

resources or assembly connectors in the PCM [275]). We select these annotations, because pattern-search-based analysis report patterns for implementation elements in the method implementations (e.g., statements or method calls) which influence data communication. However, these implementation elements do not represent the communication themselves.

To address the need for calculating the semantic relations during the coupling, the pattern-search-based coupling approach uses a central coupling model. This coupling model which captures information necessary from both analyses to enable identifying semantically related relations allowing to determine non-compliance between the vulnerabilities detected and the security characteristics annotated. Once the coupling model is created by transformations from the input and output models of the analyses, it is traversed by *Search Strategies*. We embed the creation of the coupling model and the application of the *Search Strategies* in a coupling process from the coupling design (Section 4.4).

Consequently, the focus of the pattern-search-based coupling approach is to provide techniques to identify semantically related relations between elements in the *Annotated Architecture* and the *Source Code Analysis Result*. This contrasts with the specification-based coupling approach, where the focus is to define necessary but not sufficient conditions that describe relations between the (meta)models of the analyses to be coupled because of the semantically identical relation type addressed.

This is also reflected in the structure of the presentation of the pattern-search-based coupling approach. First, we define non-compliance between the *Annotated Architecture* and the *Source Code Analysis Result* in Subsection 5.2.1. We then abstract from specific analyses and their (meta)models by presenting information necessary for the pattern-search-based coupling approach in form of reference metamodels presented in Subsection 5.2.2. Subsection 5.2.3 presents a discussion why applying the ICs of the *specification-based coupling approach* is not feasible in the pattern-search-based coupling. This discussion diverges from the general structure of the coupling approaches presented in Chapter 5. We then introduce the *Security Relations* metamodel and the coupling model which enable determining the relations necessary to perform the parameterization in Subsection 5.2.4. Again, the presentation of the coupling model deviates from the general structure of the coupling approaches presented in Chapter 5 because the specification-based coupling approach depends on the established concepts of correspondence (meta)models. The specialization of the coupling process for the pattern-search-based coupling approach is then presented in Subsection 5.2.5. Finally, we discuss the effort arising when applying the pattern-search-based coupling approach in Subsection 5.2.6.

5.2.1. Defining Non-Compliance for the Pattern-Search-Based Coupling

The goal of the pattern-search-based coupling approach is to detect architectural vulnerabilities arising from non-compliance which is caused by data that is required to be encrypted according to the *Annotated Architecture* is not encrypted in the implementation due to vulnerabilities. For this purpose, we now define when non-compliance between

security characteristics specifying data encryption in the *Annotated Architecture* and values resulting from vulnerabilities in the implementation occurs.

For detecting this non-compliance, we rely on analyses that search for patterns indicating insecure application of encryption in the implementation. A reported pattern indicates that the source code analysis detected a structure in the implementation which implies a vulnerability (see Subsubsection 2.4.2.3). Examples for such patterns that indicate insecure data encryption are the usage of weak encryption algorithms (e.g., *DES*) or the misconfiguration of the communication (e.g., use of *HTTP* instead of *HTTPS*). In our running example, we illustrate such a pattern in partial model (d) in Figure 3.2 as the usage of the value *DES* when selecting the cipher for the encryption of data (*Cipher.getInstance("DES")*).

According to Definition 1, non-compliance arises when the value of the security characteristic in the coupling scenario specified for an architectural element cannot be mapped to the values resulting from the implementation for a corresponding element in the implementation. Similarly to the specification-based coupling, pattern-search-based analyses report about the existence of patterns rather than about the value for the security characteristic of data encryption resulting from the implementation. Consequently, we have to define the values resulting from the implementation based on the reported patterns indicating vulnerabilities before defining non-compliance.

We comprehend the report of an encryption-related pattern to indicate that data is insufficiently encrypted in the implementation. Common patterns indicate that data is not encrypted at all (e.g., use of *HTTP* instead of *HTTPS*) or that data is encrypted under insecure conditions (e.g., using a weak encryption algorithm such as *DES* or hard-coded credentials). We comprehend weak encryption as insufficient encryption because application of weak cryptography is discouraged by regulations (e.g, [21, 22, 87]), or it is described as critical vulnerabilities by the OWASP [239]. Because the result of a pattern-search-based analysis is binary (i.e., either a pattern is detected or not), coupling approaches can only make binary statements about the value of the security characteristic for data encryption resulting from the implementation: Either data is sufficiently encrypted (i.e., no pattern indicating a vulnerability is reported) or data is insufficiently encrypted (i.e., a pattern indicating a vulnerability is reported).

Interpreting the results of a pattern-search-based analysis as binary statements leads to the requirement, that we can only address security characteristics that have binary values. An example for such a security characteristic is the security characteristic *Encryption* (represented by «*encrypted*» in partial model (**a**) of Figure 3.1), specifying whether data has to be encrypted or not. For such security characteristics, we can exploit the semantically related relation type where a vulnerability *violates* a security characteristic. As security characteristics commonly state expected security properties, we interpret a reported vulnerability as the indication that the expected security property is not fulfilled in the implementation, i.e., the value of the security characteristic resulting from the implementation is *false*.

Consequently, a reported pattern in the *Source Code Analysis Result* results in non-compliance if three conditions hold. First, the pattern reported must indicate missing or insufficient data encryption. Secondly, an architectural element must be annotated with

the security characteristic of the coupling scenario specifying that data has to be encrypted. Thirdly, the data in the implementation must be communicated through implementation elements which can be mapped to the annotated architectural element. This definition is based on our requirement that we do not consider behavioral descriptions in the *Annotated Architecture* for the coupling. Consequently, the data affected by the pattern can only be addressed by *structural* elements in the *Annotated Architecture* (e.g., parameters of services, connectors implying data communication between them or communication links in the PCM [275]).

In our running example, the pattern found in the implementation affects the data *encUserData* because the cipher resulting from the selection of the weak encryption algorithm *DES* is used for the encryption. We interpret the value *false* for the encryption of this data because of the vulnerability indicated by the pattern. This data is communicated between the classes corresponding to the *OnlineShop* component and the *DataStorage* component via the method *storeData*. The annotation «*encrypted*» for the communication link *Link 1* specifies that data has to be encrypted when sending it over this link. We interpret the existence of this annotation as the value *true* for the security characteristic annotated for *Link 1*. Consequently, non-compliance occurs because the value *false* resulting from the implementation for this communicated data cannot be mapped to the value *true* specified for the *Link 1* over which the data is communicated. The definition of non-compliance highlights the main challenge in this coupling approach: The *semantic relation* between elements in the *Annotated Architecture* (e.g., security characteristics and elements indicating communication) and elements in the *Source Code Analysis Result* (i.e., patterns and elements used in method implementations) has to be established by interpretation.

5.2.2. Reference Metamodels for the Pattern-Search-Based Coupling

We now introduce the reference metamodels that define the information necessary for the pattern-search-based coupling approach. As for the specification-based coupling approach, the input and output metamodels of the analyses must conform to these reference metamodels to be applicable in our coupling approach. To determine the conformance between reference metamodels and metamodels, we use the Definition 6 introduced in Subsubsection 4.4.2.3. We also present the expected semantics of the metaclasses of the reference metamodels so that a semantic correct mapping can be determined. For every reference metamodel, the commonalities and differences with the specification-based coupling approach is discussed.

Pattern-search-based analyses do not use any specifications which can be used to establish the coupling. Consequently, in contrast to the specification-based coupling approach, we cannot define one reference metamodel for the *Annotated Architecture* and the *Source Code*. Thus, we define separate reference metamodels for all models involved in the coupling process (i.e., the *Annotated Architecture*, the *Source Code*, the *Source Code Analysis Result* and the *Parameterization Model*).

5.2.2.1. Reference Metamodel: Annotated Architecture

Detecting non-compliance in the pattern-search-based coupling approach requires the architectural analysis to use specifications of security characteristics in the *Annotated Architecture* (see Subsection 4.1.1). Consequently, we expect the *Annotated Architecture* to comprise *System Elements* and *Specification Elements*. We discuss these elements in the context of the pattern-search-based coupling approach in the following. In contrast to the specification-based coupling approach, we do not consider *Configuration Elements* (i.e., *Security Policy* or a *Configuration*). We do not consider *Configuration Elements* because pattern-search-based analyses are commonly applied to detect structures that either exist or not exist in the implementation. Consequently, what-if scenarios can only be expressed for different implementations which can be implemented as an additional execution of the coupling approach. Similarly, the pattern-search-based analyses do not use security policies because no specifications are used in the analysis.

System Elements Our coupling approach for pattern-search-based analyses is an example on how to address semantically related elements in the *Annotated Architecture* and the *Source Code Analysis Result*. To address this relation on the system level, we expect the annotation in the *Annotated Architecture* to annotate architectural elements that represent communication. These elements represent communication between services of components on different levels of granularity. An example for such architectural elements are connectors between required and provided interfaces of components (e.g., in the PCM [275] or UML component diagrams [235]) or communication links between resources (e.g., used in [12, 154, 172]).

Specification Elements As in the specification-based coupling approach (see Subsection 5.1.2), we expect the *Annotated Architecture* to contain the specification elements *Security Characteristics* and *Annotations*. We use the same definition of *Security Characteristic* and *Annotation* as in the specification-based coupling described in Subsection 5.1.2. Based on our focus on encryption for the pattern-search-based coupling approach in this thesis, we expect the *Annotated Architecture* to comprise *Security Characteristics* and *Annotations* that represent the requirement for *data encryption*. However, a pattern-search-based source code analysis only provides information on whether a *pattern exists* which identifies encryption-related vulnerabilities. We interpret this result as a binary statement: a pattern is found or not (see Subsection 5.2.1). This contrasts the specification-based coupling, where the *Source Code Analysis Result* can contain various values of the security characteristic. Based on this binary statement we can only perform binary actions for the annotations in the *Annotated Architecture*.

Security characteristics with more than two values would require the coupling expert to define actions using contextual knowledge. In *UMLSec* [154], for instance, the security of a communication link can be annotated with the values *LAN*, *Internet* or *Encrypted* to express different degrees of security. When a pattern indicating a weakness violating encryption is found, it would be necessary to define whether the communication is performed over

LAN or *Internet*. However, this information is specific to the system under analysis, which is why UMLSec provides these labels as dedicated attributes. The PCM [275] allows to define the type of a communication link (e.g., *LAN* or *Internet*), but does not enforce this specification. Consequently, in this dissertation we assume the security characteristics of the coupling scenario in the *Annotated Architecture* to be represented by a binary value range in the *Annotated Architecture*: communicated data over the communication is expected to be encrypted or is not expected to be encrypted. This binary value range can be represented by an annotation specifying a binary value such as true or false or by the presence or absence of an annotation (e.g., «*encrypted*» in our running example illustrated in partial model **(a)** of Figure 3.1). We are aware that this assumption limits the applicable architectural analyses as further discussed in Section 5.3.

5.2.2.2. Reference Metamodel: *Source Code*

In contrast to the specification-based coupling, a pattern-search-based source code analysis *does not use specifications* of security characteristics to investigate whether a vulnerability exists. Thus, the reference metamodel for the *Source Code* in the pattern-search-based coupling only comprises *System Elements* describing the structure and behavior of the system in the implementation. We specifically expect fields in classes and statements, method invocations and field accesses in the method implementation as *System Elements* as they are often investigated in the patterns that search for encryption-related issues. An example is the *System Element* representing the statement invoking the method *Cipher.getInstance* with the parameter value *DES* in our running example (illustrated in partial model (d) in Figure 3.2). Elements for providing specifications can exist in the *Source Code* but are not relevant for our coupling.

We do not provide a metaclass representing patterns in the *Source Code*. While many pattern-search-based analyses allow to provide the patterns as analysis input (e.g., in CodeQL [100, 214] or PMD [260]), we also found analyses with predefined patterns that are captured in the analyzer (e.g., Snyk [310] or SonarQube [313]). Even more importantly, the patterns used in the analysis are not yet related to the system under analysis. Thus, we cannot use the patterns used for analysis to identify the related security characteristics or the annotations in the *Annotated Architecture* that can be affected by non-compliance. Consequently, requiring the *Source Code* to provide patterns while we cannot use them to enable the parameterization would unnecessarily restrict the applicability of our coupling approach.

We already argue in the reference metamodel for the *Annotated Architecture* that we do not consider *Configuration Elements*. The same reasons apply to the *Source Code*. Thus, we also do not capture a *Security Policy* metaclass or *Configuration* metaclass in the *Source Code*.

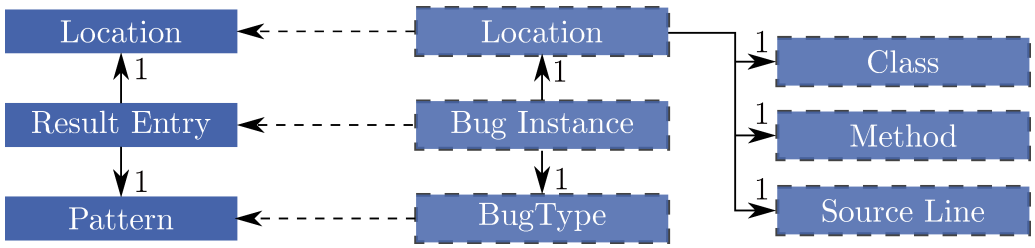


Figure 5.5.: Metaclasses and Reference of the reference metamodel for *Source Code Analysis Result* (left), and metaclasses and references for *Find Security Bugs* in the running example in Section 3.2 (right). Please see Figure 5.2 for the legend.

5.2.2.3. Reference Metamodel: *Source Code Analysis Result*

The *Source Code Analysis Result* reports locations in the implementation where patterns are found (see Section 2.4). We capture this information with reference metamodel of the *Source Code Analysis Result* for the pattern-search-based coupling as illustrated in Figure 5.5. Our reference metamodel defines *Result Entries* $re \in RE$, which describe the occurrence of a pattern at a specific location in the source code. The *Result Entry* references the metaclasses *Location* $l \in L$ in a set of all locations L and a *Pattern* $p \in P$ from the set of all patterns P (described in Equation 5.8). *Location* and *Patterns* are common provided information by pattern-search-based source code analyses such as *Find Security Bugs* [18], PMD [260], SonarQube [313] or Snyk [310].

$$\forall re \in RE\colon \exists l \in L, \exists p \in P\colon \langle re, l\rangle \wedge \langle re, p\rangle \tag{5.8}$$

Location. The *Location* metaclass is an abstract representation of a *System Element* in the *Source Code*. We do not reference the *System Element* of the *Source Code* directly to highlight that the *Location* in the *Source Code Analysis Result* is often used to represent the *position* in the source code the pattern is found. This approach is reasonable because it enables Integrated Development Environments (IDEs) to highlight this location or to navigate the source code automatically.

Pattern. The *Pattern* metaclass identifies the pattern found in the implementation that indicates a weakness in the system. We expect the *Pattern* to be identified by a name or an ID specific to the analysis. In our coupling approach, we focus on patterns that are related to a weakness violating encryption.

We do not expect the *Source Code Analysis Result* to comprise the weakness associated to the pattern. We made this decision deliberately because we found that not all analyses provide the association between pattern and the respective weaknesses in the *Source Code Analysis Result*. For example, *Find Security Bugs* [18] uses a dedicated document such as a website or catalog in the analysis tooling to establish this association. However, such sources can be difficult to parse automatically or cannot be accessed during the coupling process.

Defining a reference metamodel for the *Source Code Analysis Result* without *weaknesses* does not limit the applicability of our coupling approach. As discussed in Subsubsection 4.4.2.3,

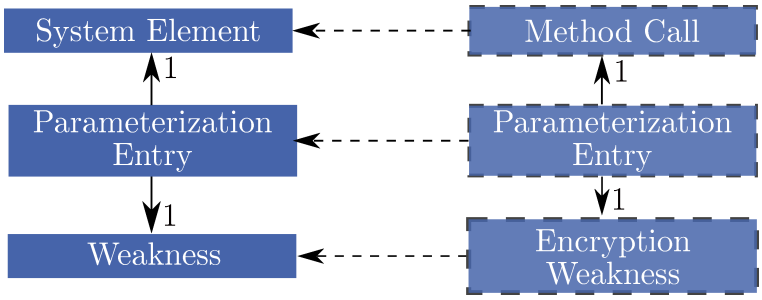


Figure 5.6.: Metaclasses and Reference of the reference metamodel for *Parameterization Model* (left), and metaclasses and references for an example metamodel of the *Parameterization Model* for *Find Security Bugs* in the running example in Section 3.2 (right). Please see Figure 5.2 for the legend.

we expect the reference metamodels to be under-specified. Thus, a source code analysis is *allowed* to provide the weaknesses in the *Source Code Analysis Result* but it is not required to do so.

5.2.2.4. Reference Metamodel: Parameterization Model

For performing the parameterization, relations between the information in the *Source Code Analysis Result* and the *Annotated Architecture* have to be determined. This includes (1) relations between the architectural elements annotated with the security characteristic of the coupling scenario and the locations reported in the *Source Code Analysis Result*, and (2) relations between security characteristic of the coupling scenario specified in the *Annotated Architecture*, and the patterns found in the implementation.

We found two challenges for the *coupling expert* arising from the information in the *Source Code Analysis Result*. First, the *Source Code Analysis Result* represents the locations with information and syntax focused on source code navigation. This representation requires the coupling expert to have expertise in parsing the syntax and mapping the comprised information to the source code elements. Secondly, we address analyses that provide the patterns found in the implementation but not the associated weaknesses. Even more, many analyses provide their own identifiers for their patterns (e.g., names or IDs). Consequently, the *coupling expert* has to understand the identification of the patterns and the relations between patterns and weaknesses. Both challenges require expertise only in the source code analysis. However, the coupling expert is expected to have knowledge about the relation *between* the architectural analysis and source code analysis rather than to have *detailed* expertise in a single analysis. In contrast, the analysis architect knows the syntax used in the *Source Code Analysis Result*, the relation between the locations and the *Source Code*, and the relation between the patterns and weaknesses. Therefore, the challenges arising from the *Source Code Analysis Result* for the coupling expert can be addressed by the analysis architect.

We support the differences in expertise by the *Parameterization Model*, which forms an interface between the coupling expert and the analysis architect (see Subsubsection 4.4.2.2).

The reference metamodel for the *Parameterization Model* contains the metaclass Parameterization Entry $pe \in PE$ which represents a weakness associated with the pattern found for an implementation element (illustrated in Figure 5.6). The *Parameterization Entry pe* references the metaclass *System Element* $\delta_{PM} \in \Delta_{PM}$ and the metaclass *Weakness* $w \in W$ from the set of all captured weaknesses W as described in Equation 5.9.

$$\forall pe \in PE\colon \exists \delta_{PM} \in \Delta_{PM} \exists w \in W\colon \langle pe, \delta_{PM} \rangle \wedge \langle pe, w \rangle \tag{5.9}$$

The *System Element* represents the element in the *Source Code* which is affected by the weakness. This information allows the coupling expert to determine the relation between the system elements in the *Annotated Architecture* annotated with the security characteristic of the coupling scenario and the elements of the *Source Code*. The *Weakness* metaclass represents the weakness found in the implementation element. Weaknesses and security characteristics are both located in the security domain. In contrast, the patterns are specific to the analysis tooling and located in the system domain. Thus, the weakness in the *Parameterization Model* allows the coupling expert to focus on a single domain (security) for establishing the relation to the security characteristic in the *Annotated Architecture* rather than having to consider multiple domains.

5.2.3. Discussion: Inapplicability of Integration Conditions

In contrast to the specification-based coupling in Subsection 5.1.3, we do not have to ensure that values resulting from the implementation are specific to a system under analysis and can be integrated into the *Annotated Architecture*. This is because we constrain the applicable security characteristics in the coupling scenario to binary value ranges. Thus, the resolution of the annotation affected by the non-compliance in the *Annotated Architecture* is reduced to calculating the security characteristic violated by a pattern and the architectural element affected by the pattern.

Additionally, the ICs in the specification-based coupling are designed to exploit the semantically identical relation type between elements of the input and output (meta)models of the architectural analysis and *specification-based* source code analysis. We formulated the ICs so that non-compliance between the *Source Code Analysis Result* and the *Annotated Source Code* also indicates non-compliance in the *Annotated Architecture*. For this, the ICs of the specification-based coupling exploit the premise that the architectural analysis and the source code analysis both use specifications of semantically identical information such as the security characteristics in the coupling scenario and their respective annotations. The ICs in the specification-based coupling additionally ensure that (1) the values resulting from the implementation can be mapped to the values of the security characteristic in the coupling scenario in the *Annotated Architecture*, and (2) that the affected annotation can be calculated. In contrast, we constrain the applicable security characteristics in the coupling scenario to binary value ranges in the pattern-search-based coupling. Consequently, we do not have to ensure that the values resulting from the implementation exist in the *Annotated Architecture*. However, we have to ensure that the semantically related

between the elements of the *Parameterization Model* and the *Annotated Architecture exist* to determine non-compliance.

Furthermore, the pattern-search-based source code analyses do not use specifications which differs from those in the specification-based coupling. Consequently, all ICs establish correspondences between elements with a relation to security characteristics over the *Source Code* cannot be applied (i.e., **IC1**, **IC3**, **IC4**, **IC5**, **IC6**, **IC9**, **IC10**). Secondly, in the specification-based coupling approach, we exploit that non-compliance arises for system elements that have a semantically identical relation to the architectural elements annotated with the security characteristic in the coupling scenario (e.g., parameters). However, such a semantically identical relation cannot be specified for the system elements in the pattern-search-based coupling approach, because the locations reported by pattern-search-based analyses refer to elements in the implementation of methods for which we do not consider corresponding architectural elements in our coupling approach. Consequently, all ICs with relation to system elements are not applicable (i.e., **IC2**, **IC3**, **IC5**, **IC6**, **IC8**). Similarly, pattern-search-based analyses do not use any configuration or security policy to perform the analysis, which results in inapplicability of the ICs involving these elements (i.e., **IC2**, **IC3**, **IC4**, **IC7**, **IC8**, **IC10**).

In summary, ICs from the specification-based coupling cannot be applied to the pattern-search-based coupling approach due to the differences in the reference metamodels, and the missing semantically identical relations between the *Annotated Architecture* and the *Parameterization Model*. Additionally, we have to ensure that the semantically related relations between the elements in the *Annotated Architecture* and the *Parameterization Model* relevant for the coupling can be determined. Finally, the pattern-search-based coupling approach requires techniques to calculate the (potentially transitive) semantically related relations between the elements in the *Annotated Architecture* and the *Parameterization Model*. Consequently, we focus on the techniques to determine the semantic relations and omit ICs for the pattern-search-based coupling approach because they would represent the semantic relations calculated by these techniques.

5.2.4. Metamodels For Establishing Relations in the Pattern-Search-Based Coupling Approach

In this section, we introduce metamodels that enable the calculation of the relations between semantically related elements in the *Parameterization Model* and *Annotated Architecture* necessary for the parameterization. We have to establish these relations on the system level and the security level. Establishing the relations on the system level is necessary to determine the annotated architectural elements affected by non-compliance. Establishing the relations on the security level is necessary to evaluate whether a pattern found in the implementation results in the violation of the values of the security characteristic of the coupling scenario specified in the *Annotated Architecture*. We establish these relations based on the hypothesis that metamodels can be used to calculate the semantic relations between syntactic elements (see Subsection 4.3.2). Based

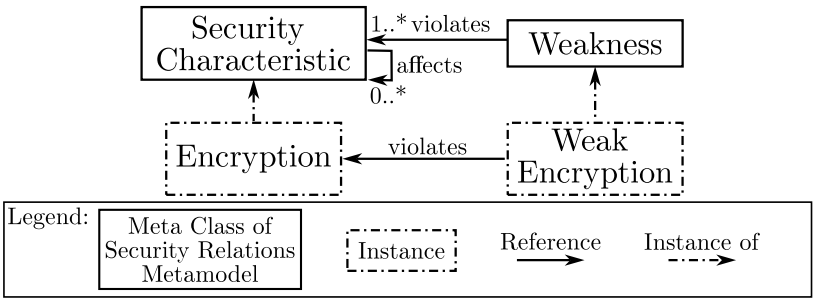


Figure 5.7.: The *Security Relations* metamodel and an instance for the running example.

on this hypothesis, we describe two metamodels in the following: The *Security Relations* metamodel and the *Coupling Metamodel*.

5.2.4.1. Security Relations Metamodel

The *Security Relations* metamodel captures the relations between *weaknesses* reported by the source code analysis and the security characteristics specified in the *Annotated Architecture* (illustrated in Figure 5.7). In the remainder of this thesis, we call the instance of this metamodel the *Security Relations* model.

In the *Parameterization Model*, we abstract the patterns in the *Source Code Analysis Result* to the weaknesses they imply. With this approach, the coupling expert can focus on the security domain rather than understanding the source code patterns specified for the implementation domain. These weaknesses are still related to the source code analyses while the security characteristics are applied in the architectural analysis. For providing an automated coupling approach, we make the relation between security characteristics and weaknesses explicit.

For this purpose, the *Security Relations* metamodel contains the two metaclasses *Security Characteristic* and *Weakness*. The *Weakness* describes a detectable weakness in the implementation. Please note that we use the term *weakness* rather than *vulnerability* because we do not refer to issue that is already found in the implementation (see Subsection 2.3.1). The *Security Characteristic* describes security-related properties in conformance of the definition in Section 2.3 (e.g., *encryption*). We capture the relation between these two metaclasses with the reference *violates*, which defines that a *Weakness* violates one or more *Security Characteristics*. The CWE [327] contains similar relations. However, the *Weaknesses* in the CWE are related to high-level security objectives such as confidentiality, integrity or availability rather than to specific security characteristics. It is important to note that the *Security Characteristic* and *Weakness* are metaclasses of our metamodel and not metaclasses or instances of specific (meta)models of the analyses. These metaclasses are used to present *concepts* like *encryption* and *weak encryption* respectively in the running example.

We do not include a metaclass *Pattern* which captures the patterns reported by the source code analyses and a relation to the weaknesses they indicate. We made this decision

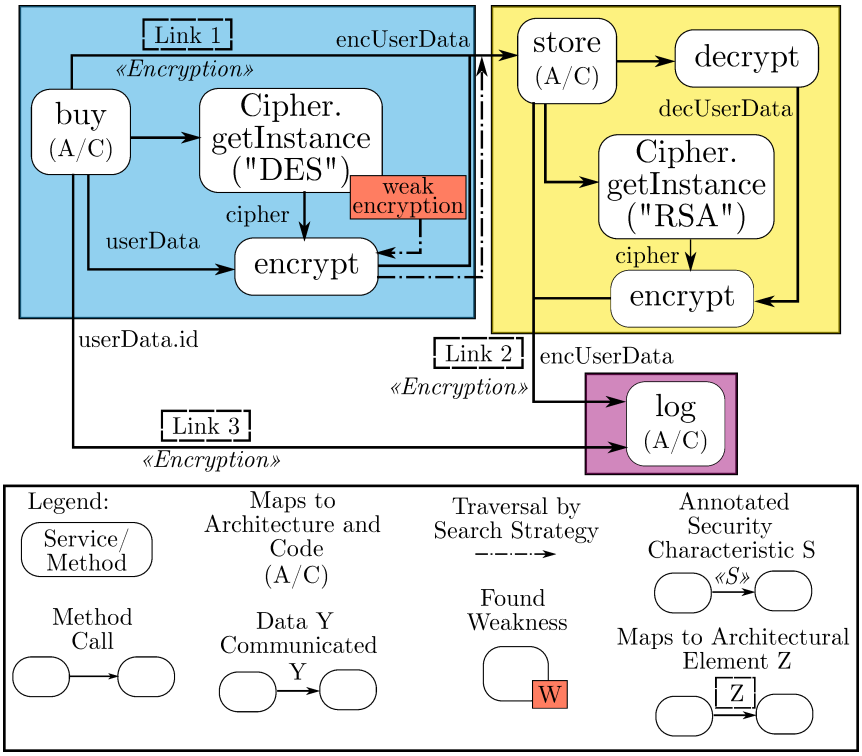


Figure 5.8.: Coupling model for running example. Only nodes mapping to the architectural model and code are annotated with (A/C). Colored boxes mark the components and corresponding classes from the running example illustrated in Figure 3.1 and Figure 3.2: OnlineShop (teal), DataStorage (yellow), and Logger (purple).

because including the *Pattern* metaclass would require the *Security Relations* model to be created by the analysis architect *and* the coupling expert. This necessity would increase the effort for creating and maintaining the *Security Relations* model. Additionally, we decided not to include the *Pattern* metaclass in the *Security Relations* metamodel to limit model complexity.

Figure 5.7 also shows an instance of the *Security Relations* metamodel for our running example, containing the *Weakness weak encryption* that violates the *Security Characteristic encryption*. We are aware that *weak encryption* is a broad weakness and could be refined to more fine-grained weaknesses. This is similar to the weakness *CWE-327: Use of a Broken or Risky Cryptographic Algorithm* in the CWE which can be refined to *CWE-328: Use of Weak Hash* and *CWE-1240: Use of a Cryptographic Primitive with a Risky Implementation*. However, the purpose of the model is to establish a semantic relation between the weaknesses and the security characteristic *encryption*. This is why the concrete instances of weaknesses can all be mapped to *weak encryption* without losing the capabilities for the coupling.

5.2.4.2. Coupling Metamodel

The purpose of the coupling metamodel is to enable relating the weaknesses detected in the implementation to the annotations of the security characteristics in the *Annotated Architecture*. Thus, the coupling metamodel builds the foundation for performing the parameterization.

For this purpose, the coupling metamodel contains elements which allows the coupling expert to determine non-compliance and identify the annotations affected by it. Figure 5.8 illustrates an instance of the coupling metamodel for our running example. The coupling metamodel has to capture the system information and the security-relevant information necessary for the coupling to which elements in the *Annotated Architecture* and *Parameterization Model* can be mapped. We first detail how the system is represented in the coupling metamodel. Then we discuss how to represent the security-relevant information. Please note that the coupling metamodel described in this section is specific to our pattern-search-based coupling approach which forms an example for investigating non-compliance regarding security characteristics describing the need for data encryption. Investigating non-compliance for other security characteristics can result in the need for another coupling metamodel.

Representing System Information In the pattern-search-based coupling approach, architectural elements in the *Annotated Architecture* which represent the communication of data are annotated to state the requirement that communicated data must be encrypted (see Subsection 5.2.2). In contrast, the patterns reported in the *Source Code Analysis Result* describe the location in the implementation involved in a vulnerability invalidating the encryption of data. We abstract the information in the *Source Code Analysis Result* in our *Parameterization Model*. These elements must allow establishing relations between the *Annotated Architecture* and the *Parameterization Model* necessary for the coupling.

Thus, the coupling metamodel has to provide the following elements: First, elements which can be mapped to methods, method calls and statements in the implementation. Secondly, elements which can be mapped to the communication represented by the architectural elements in the *Annotated Architecture*. Thirdly, elements to which the services provided by components can be mapped, which allow to identify the communication based on the methods or services involved in the communication. We propose that the information that is communicated is captured by the coupling metamodel. This information allows increasing precision of the coupling because the encryption weaknesses affect the information communicated.

We use a *Call Graph*-like coupling metamodel with data flow information to capture the prior elements for illustration of our coupling approach in this chapter. The metamodel contains the metaclasses *Node* and *Edge*. *Node* can be mapped (1) to callable methods that correspond to services provided by components, and (2) to statements and method calls in the implementation without correspondence to the architecture. The second property is essential because locations in the *Source Code Analysis Result* often refer to statements

or method calls for which no corresponding elements in the *Annotated Architecture* are considered for the coupling (see Subsection 4.1.1). Examples are calls to library methods or selection of a certain encryption algorithm. Nodes of the coupling metamodel in Figure 5.8 include *buy*, *store*, and *Cipher.getInstance("DES")*. *Edges* capture communication between these *Nodes*. On the architectural level, this communication can capture communication links between resources or connectors between services. On the implementation level, this communication is the result of method calls and the data flow induced, amongst others, by statements and the return of a method. Edges of the coupling metamodel in Figure 5.8 represent the communication between the *buy* and *store* as well as the call of *Cipher.getInstance("DES")* from *buy*.

For the coupling, it is important to differentiate between edges that can be mapped to the *Annotated Architecture*, and those that cannot. Only those edges that can be mapped to the *Annotated Architecture* can be used for parameterization because only they can be annotated with the security characteristic of the coupling scenario. Thus, we differentiate the instances of *Edges* regarding their visibility in the *Annotated Architecture* into *visible edges* and *hidden edges*. A *visible edge* connects nodes that can be mapped to corresponding services (*Annotated Architecture*) *and* methods (*Source Code*). A *hidden edge* connects nodes where one or both nodes connected by the edge cannot be mapped to the *Annotated Architecture*. In Figure 5.8, the edge between the node *buy* and the node *store* is a *visible edge* as it connects two nodes described in the *Annotated Architecture* and *Source Code*. In contrast, the edge between the node *buy* and the node *Cipher.getInstance("DES")* is a *hidden edge* as the latter is not described in the *Annotated Architecture*.

When the system is described by a joint code base, we propose to extract both types of edges from the implementation. In the prototypical implementation of the coupling [269], we extract a call graph with the tool SPOON [249] and transform the edges in the call-graph to edges in the coupling metamodel. This approach cannot be applied if the system is described by multiple programs without a joint code base. In this case, the *visible edges* must be extracted from the *Annotated Architecture* or from additional information of the implementation, such as configuration files (e.g., as performed by Kirschner et al. [164]). The *hidden edges* must be extracted from the *Source Code* of each program. The extracted information has then to be joined in the coupling metamodel.

The discussed concepts are also transferable to other metamodels like the GRaViTY program model [250]. Call graphs without data flow information are another option for the coupling metamodel. However, the lack of information about the data communicated can reduce the precision of the coupling approach as the data flow — and consequently the mappings to the *Annotated Architecture*— must be approximated by heuristics. We show in the bachelor's thesis of Traub [336] that the selection of the coupling metamodel can influence the precision of a pattern-search-based coupling approach by comparing the application of call graphs to the usage of system dependence graphs [347].

Representing Security-Relevant Information Detecting non-compliance requires relating the patterns detected to annotated security characteristics in the *Annotated Architecture*.

We abstract the detected patterns which are represented in the *Source Code Analysis Result* with the *Parameterization Model*. This abstraction contains weaknesses that violate certain security characteristics as described in the *Security Relations* metamodel. Thus, we include the *Security Characteristics* and *Weaknesses* of our *Security Relations* metamodel in the coupling metamodel. For this purpose, the metaclass *Node* of the coupling metamodel references the metaclass *Weakness* of the *Security Relations* model. This allows to annotate *Nodes* with *Weaknesses* of a *Security Relations* model, implying that a vulnerability conforming to this weakness is found in the implementation elements represented by the node. The metaclass *Edge* of the coupling metamodel references the metaclass *Security Characteristics* of the *Security Relations* metamodel. This allows to annotate *Edges* with *Security Characteristics* of a *Security Relations* model, describing that the data communicated over this edge should conform to the security characteristic (i.e., data should be encrypted in our coupling scenario).

5.2.5. Specializing the Coupling Process For Pattern-Search-Based Analysis Coupling

We now detail the specialization of the coupling process (presented in Section 4.4) for coupling architectural analysis with pattern-search-based source code analysis. The resulting coupling process is illustrated in Figure 5.9. In addition to the process steps outlined in Section 4.4, we introduce an additional process step *Security Relations Model Definition* which is specific to the pattern-search-based coupling.

5.2.5.1. Security Relations Model Definition

The relation between instances of *Security Characteristic* and instances of *Weaknesses* described in the *Security Relations* models allows determining if a weakness violates a security characteristic. From this relation, non-compliance between security characteristics specified in the *Annotated Architecture* and patterns found in the *Source Code* can be derived. Thus, we introduce the *Security Relations Model Definition* as a process step. In this step, the coupling expert has to create a *Security Relations* model or select an existing one. We realize this in the transformation *Security Relations Model Creation* as illustrated in Figure 5.9. This transformation has no input models and provides the *Security Relations* model as output. This model has to capture *Security Characteristics* which map to the security characteristic in the coupling scenario and *Weaknesses* that violate them. Figure 5.7 illustrates the *Security Relations* model for our running example, which captures the *Weakness weak encryption* violating the *Security Characteristic encryption*.

The *Security Relations Model Definition* process step is not included in the coupling process of the coupling design (see Subsection 4.4.3) because it is specific to the pattern-search-based coupling. This step is executed *prior* to the remaining coupling process as the *Security Relations* model is used already in the *Analyses Input Model Consistency* as illustrated in Figure 5.9. Another reason for performing this step prior to the remaining process is the

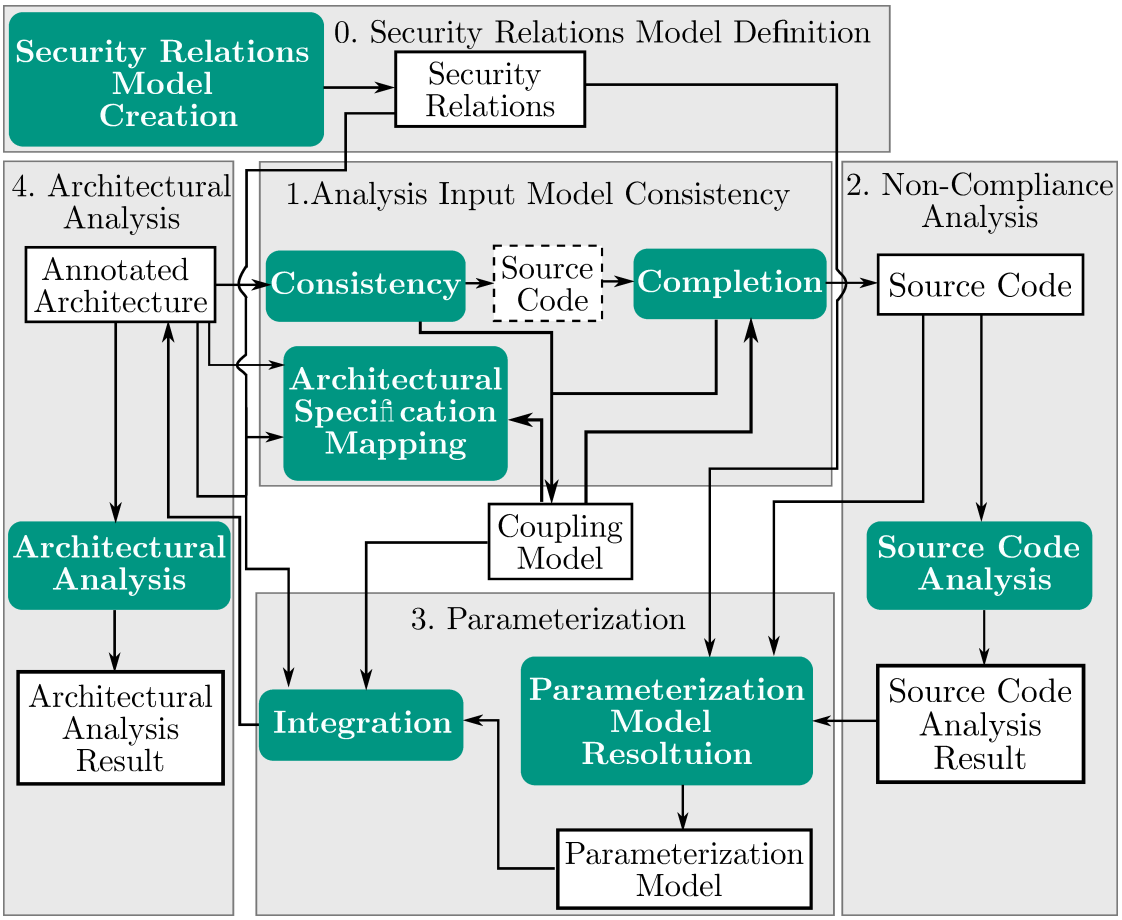


Figure 5.9.: Models and process steps of the coupling process extended for the pattern-search-based coupling approach. Sequential increasing numbers order the execution of the coupling process steps. For legend please see Figure 4.2.

need for finding a pattern-search-based source code analysis that detects patterns violating the weaknesses. However, the relation between pattern and weakness is not clear for every analysis (as discussed in Subsection 5.2.2). Consequently, the coupling expert interacts with the respective analysis architect to determine whether the pattern-search-based source code analysis is able to detect patterns relating to the *Weaknesses* in the *Security Relations* model.

5.2.5.2. Analyses Input Model Consistency

The *Security Relations* model is used as input for the *Analyses Input Model Consistency* step in addition to the *Annotated Architecture*, and the *Source Code* prior to the evolution step. The output of the *Analyses Input Model Consistency* step is an incomplete coupling model and the *Source Code* that is consistent with the *Annotated Architecture*. Similar to Figure 4.2, we simplify Figure 5.9 by not illustrating the use of the *Source Code* before performing the evolution step. We adapt the *Consistency* transformation and *Completion* transformation of the *Analyses Input Model Consistency* step to realize the pattern-search-based coupling

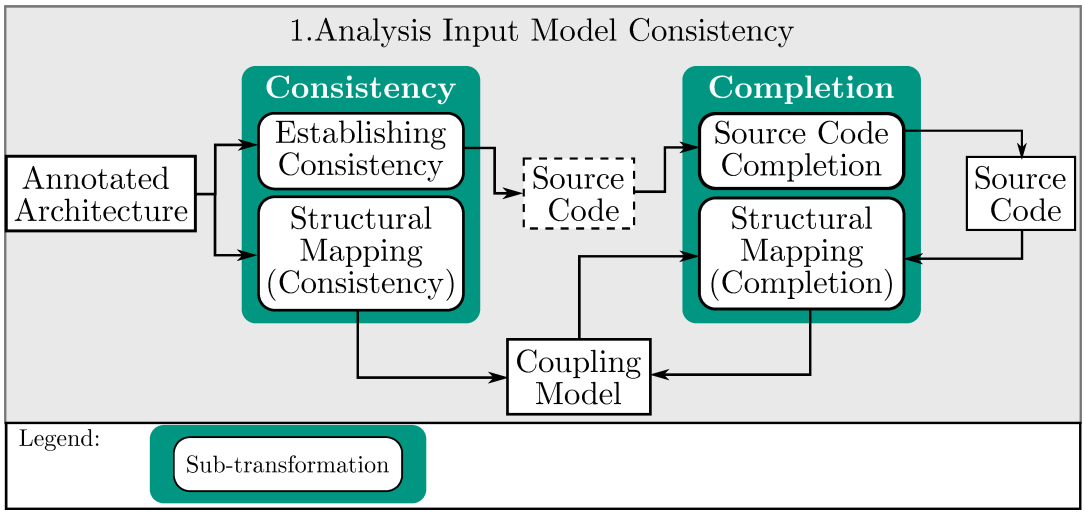


Figure 5.10.: The sub-transformations of the *Consistency* transformation and the *Completion* transformation, including their models used, and the flows between sub-transformations and models. See Figure 4.2 for elements not represented in legend.

process. In addition, we introduce the *Architectural Specification Mapping* transformation to annotate the coupling model with the security characteristic in the scenario.

Consistency Transformation The *Consistency* transformation uses the *Annotated Architecture* and the *Source Code* before performing the evolution step and provides the *incomplete Source Code* and a coupling model. The *Consistency* transformation performs two tasks. First, the transformation establishes consistency between the *Annotated Architecture* and the *Source Code*. Secondly, the transformation creates a coupling model which captures nodes and edges for which mappings to the *Annotated Architecture and* the *Source Code* exist. We logically subdivide the *Consistency* transformation into the sub-transformations *Establishing Consistency* and *Structural Mapping (Consistency)* as illustrated in Figure 5.10. The *Establishing Consistency* transformation establishes consistency between the *Annotated Architecture* and the *Source Code*. The *Structural Mapping (Consistency)* transformation creates a coupling model. This division highlights that both tasks can be performed independently and enables the coupling expert to replace the coupling metamodel without the need to modify the *Establishing Consistency* transformation. However, both tasks can also be performed in a single transformation to avoid redundant model traversals.

Establishing Consistency. The *Establishing Consistency* transformation uses the *Annotated Architecture* and the *Source Code* before the evolution step as input, and provides the *incomplete Source Code* as output. The *incomplete Source Code* is consistent with the *Annotated Architecture* regarding the services provided by components and the communication implied by the architectural elements. In our running example, the *Establishing Consistency* transformation ensures, amongst others, that the *Source Code* contains methods *buy*, *store* and *log* corresponding to the services provided by the components in the *Annotated Architecture*. In addition, the *Establishing Consistency* transformation ensures that the *Source Code* contains no method calls that are not implied by an architectural element

in the *Annotated Architecture*. This case cannot arise in our running example because all resources are connected by a communication link. However, if the communication link *Link2* does not exist, the transformation should remove the call of the *log* method from the *store method* or notify a software engineer about this inconsistency.

Structural Mapping (Consistency). The *Structural Mapping (Consistency)* transformation uses the *Annotated Architecture* as input and provides a coupling model as output. This transformation first creates nodes for services in the *Annotated Architecture* and methods in the *Source Code* for which consistency has been established in the *Establishing Consistency* sub-transformation. As illustrated in Figure 5.8, three nodes are generated in our running example for the methods and their corresponding services: *buy*, *store* and *log*. In addition, the *Structural Mapping (Consistency)* creates the *visible edges* between these nodes, capturing communication between services and their corresponding methods. For these edges, we consider that one architectural element representing communication can map to multiple edges. A communication link between resources has to be mapped to all edges between nodes which also represent the communication between services over the corresponding link. For our running example three *visible edges* are generated: the edge between *buy* and *store* nodes, the edge between the *buy* and *log* nodes and the edge between the *store* and *log* nodes. These edges are generated because the communication is performed over the communication links *Link1*, *Link2* and *Link3* respectively. Figure 5.8 illustrates this mapping by annotating the *visible edges* with the names of the respective communication links.

Architectural Specification Mapping Transformation The *Architectural Specification Mapping* transformation uses the *Annotated Architecture*, the *Security Relations* model and the coupling model from the *Structural Mapping (Consistency)* transformation as input and provides a coupling model as output. The coupling model used as input is enriched with representations of the security characteristics in the coupling scenario annotated in the *Annotated Architecture*.

In our coupling approach, the security characteristic of the coupling scenario is annotated to architectural elements representing communication. Thus, the *Architectural Specification Mapping* transformation annotates all *visible edges* of the coupling model which correspond to an annotated architectural element. These *visible edges* are annotated with Security Characteristics of the *Security Relations* model that map to the security characteristic of the coupling scenario. If the *Architectural Specification Mapping* transformation is performed before the *Completion* transformation, the coupling model only contains *visible edges*. Consequently, the transformation can inspect for each edge in the coupling model whether the corresponding architectural element in the *Annotated Architecture* is annotated with the security characteristic in the coupling scenario. If a communication link in our running example is annotated with «*encrypted*», all *visible edges* that map to this link must be annotated with *encryption*. Figure 5.8 illustrates this mapping by annotating the *visible edges* between the nodes *buy* and *store*, *buy* and *log* as well as *store* and *log* with «*encrypted*», because they map to the communication links *Link1*, *Link2* and *Link3* respectively. This

example can also transfer to other types of architectural elements implying communication, such as assembly connectors between services (e.g., in UML [235] or the PCM [275]).

Completion Transformation The *Completion* transformation uses the incomplete *Source Code* and the coupling model after the *Architectural Specification Mapping* transformation as input, and provides a completed *Source Code* and a coupling model as output.

Similar to the specification-based coupling approach, we expect that the incomplete *Source Code* does not contain all implementation elements in the evolution of the implementation that are relevant for the analysis. Examples for such elements are method bodies of new services in the evolution step, modifications of existing methods or new fields. In the initial development phase, the incomplete *Source Code* would only contain method signatures and empty method bodies. This is especially relevant as encryption weaknesses arise, for example, by selecting weak encryption algorithms in encryption functionality. This selection is exemplified by the method call *Cipher.getInstance("DES")* in the implementation of the *buy* method in our running example (illustrated in partial model (d) of Figure 3.2). Thus, the incomplete *Source Code* cannot be analyzed with a pattern-search-based analysis to detect patterns that indicate encryption-related vulnerabilities.

The coupling model also cannot be used in the analysis coupling, because it misses nodes and edges representing the behavior of the services in the architecture which correspond to methods in the implementation. Consequently, missing those nodes and edges renders the coupling model unusable because the patterns reported for the locations in the *Source Code Analysis Result* cannot be represented. In the coupling model after the *Architectural Specification Mapping* transformation, for example, the node representing the *Cipher.getInstance("DES")* method call in Figure 5.8 is missing. To address these missing elements, the *Completion* transformation has to perform two tasks: (1) completing the *Source Code* to the final implementation and (2) modifying the coupling model to add nodes and edges resulting from the completed *Source Code* that cannot be mapped to the *Annotated Architecture*. Similar to the *Consistency* transformation, we logically subdivide the *Completion* transformation into the two sub-transformations *Source Code Completion* and *Structural Mapping (Completion)* as illustrated in Figure 5.10. This division again enables the coupling expert to replace the coupling metamodel without the need to modify the *Source Code Completion* transformation. However, both tasks can also be performed in a single transformation to avoid redundant model traversals.

Source Code Completion. The *Source Code Completion* transformation uses the incomplete *Source Code* as input and provides the *Source Code* as output. In the *Source Code Completion* transformation, the *Source Code* is completed to the final implementation. Similar to the *specification-based* coupling, this includes tasks like providing method bodies, fields or methods without correspondence to the *Annotated Architecture*. In contrast to the *specification-based* coupling, we do not expect the *Source Code Completion* transformation to provide annotations in the *Source Code* for the purpose of the coupling.

Structural Mapping (Completion). The *Structural Mapping (Completion)* transformation uses the completed *Source Code* and the coupling model after the *Architectural Specification*

Mapping transformation as input and provides the same coupling model as output. This transformation inserts nodes and *hidden edges* into the coupling model that do not map to the *Annotated Architecture*. When implementation elements that should map to nodes are not represented in the coupling model, new nodes are created. These nodes represent implementation elements such as statements, method calls or usage of fields that are relevant for the analysis coupling. Then the communication between all implementation elements in the *Source Code* which are mapped to the coupling model is investigated. For each communication not captured, the *Structural Mapping (Completion)* transformation creates edges between the corresponding nodes. The inserted edges and nodes have then to be mapped to the nodes and edges existing due to the *Structural Mapping (Consistency)* transformation. However, a method in the *Source Code* without correspondence to a service in the *Annotated Architecture* can call a method in the *Source Code* corresponding to a service in the *Annotated Architecture*. Such communication is represented by a *hidden edge* in the coupling model. An example for such a case in our running example would be if the *encrypt* method called by the *buy* method would itself call the *log* method. Inserting the *hidden edges* without further actions would result in losing the information about which communication should be encrypted. In the example given above, the information that the communication must be encrypted because it is performed over *Link3* would be lost for the respective *hidden edge*. To address this challenge, the security characteristics provided to the *visible edges* have to be transferred to the *hidden edges* if they are both connected to one common node.

5.2.5.3. Non-Compliance Analysis

The *Non-Compliance Analysis* step uses the *Source Code* as input and performs the pattern-search-based source code analysis in the *Source Code Analysis* transformation. The *Non-Compliance Analysis* step provides the *Source Code Analysis Result* as output that contains patterns and their found locations that may result in non-compliance with the *Annotated Architecture*.

Source code analyses like CodeQL [100, 214] or PMD [260] allow specifying custom patterns to be searched in the source code. This requires selecting the patterns that relate to the weaknesses in the *Security Relations* model that violate the security characteristic in the coupling scenario. If an analysis allows this selection, the coupling expert has to ensure to select *all* patterns that relate to the relevant weaknesses. Otherwise, the analysis coupling may miss non-compliance between the implementation and the *Annotated Architecture*.

5.2.5.4. Parameterization

The *Parameterization* step performs the parameterization by determining non-compliance between the detected patterns in the *Source Code Analysis Result* with the security characteristics of the coupling scenario specified in the *Annotated Architecture*. For this parameterization, the *Parameterization* step uses the *Source Code Analysis Result*, the coupling model after the *Completion* transformation, the *Source Code*, the *Security Relations* model and

the *Annotated Architecture* as input, and provides a parameterized *Annotated Architecture* as output. We realize the *Parameterization* step by the *Parameterization Model Resolution* transformation and the *Integration* transformation.

Parameterization Model Resolution The *Parameterization Model Resolution* uses the *Source Code*, the *Source Code Analysis Result* and the *Security Relations* model as input and provides the *Parameterization Model* as output.

This transformation creates the *Parameterization Model* to address the challenges for the coupling expert arising from the locations and patterns provided in the *Source Code Analysis Result* and from its syntax (see Subsection 5.2.2). For every *Result Entry* in the *Source Code Analysis Result*, the *Parameterization Model Resolution* transformation has to perform two tasks to provide a *Parameterization Entry*. First, the patterns in the *Source Code Analysis Result* have to be transformed to the instances of *Weaknesses* in the *Security Relations* model. Second, the reported location has to be transformed to an element of the *Source Code*.

The first task can be solved by the coupling expert if documentation about the relation between a pattern and a weakness is available. Such documentation is provided for *Find Security Bugs* [18] as a website mapping patterns to IDs of the CWE. However, as a worst-case scenario, we assume a documentation not to be complete or not available. For example, the analysis tool *Flawfinder* [354] does not document IDs of the CWE for all patterns or whether no corresponding CWE exist. Missing documentation introduces the challenge for the coupling expert to understand which security weaknesses may arise when a certain pattern is reported. The *analysis architect* knows about this relation, as otherwise the security analysis would not search the patterns reported. Thus, we expect the *analysis architect* to realize the transformation between pattern in the *Source Code Analysis Result* and the instances of *Weaknesses* in the *Security Relations* model. However, the coupling expert is aware of the weaknesses relevant for the coupling scenario as they provide the *Security Relations* model in the *Security Relations Model Definition* process step. Consequently, the analysis architect has to interact with the coupling expert to determine the instances of *Weaknesses* in the *Security Relations* model a pattern maps to. In our running example, the *analysis architect* has to provide a transformation that maps the pattern *DES_USAGE* to the instance of *Weakness weak encryption* in the *Security Relations* model.

We address the second task by parsing the location a pattern is found to provide an implementation element from the *Source Code*. This functionality has to be provided by the analysis architect as they know about the relation between the implementation element of the *Source Code* and the location provided in the *Source Code Analysis Result*. In our running example, the location of the found pattern is represented by the name of the class (*OnlineShop*), the respective method (*buy*) and a line (*5*). Thus, to calculate the corresponding implementation element, the analysis architect has to provide a transformation that finds the file containing the source code of the class *OnlineShop*, searches for the line of code, and returns the corresponding implementation element in the *Source Code*. In the

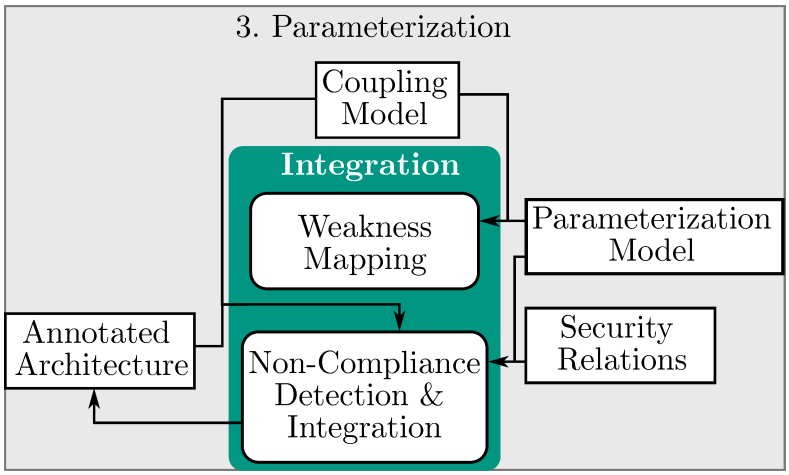


Figure 5.11.: The sub-transformations of the *Integration* transformation, the models used in these transformations, and the flows of these models. See Figure 4.2 and Figure 5.10 for legend.

running example, this transformation yields the implementation element representing *Cipher.getInstance("DES")*.

Integration In the specification-based coupling approach, no explicit non-compliance detection is necessary because every vulnerability in the *Source Code Analysis Result* directly indicates non-compliance with the *Annotated Architecture* (see Subsection 5.1.4). In contrast, the pattern-search-based coupling requires an explicit non-compliance detection, because the weaknesses found in the implementation and the security characteristics annotated in the *Annotated Architecture* are semantically related and exist on different abstraction levels. This property results in the necessity to determine whether a weakness found in the implementation actually violates a security characteristic specified in the *Annotated Architecture*. Consequently, we define the *Integration* transformation to perform three tasks in the pattern-search-based coupling approach. First, the found weaknesses represented in every *Parameterization Entry* of the *Parameterization Model* have to be mapped to the coupling model. Secondly, non-compliance has to be detected. Thirdly, if non-compliance has been detected, the annotations in the *Annotated Architecture* have to be modified. To solve these tasks, the *Integration* transformation uses the coupling model after the *Completion* transformation, the *Parameterization Model*, the *Security Relations* model and the *Annotated Architecture* as input and provides a parameterized *Annotated Architecture* as output. We divide these tasks into two logical sub-transformations *Weakness Mapping* and *Non-Compliance Detection & Integration* as illustrated in Figure 5.11.

Weakness Mapping. In the *Weakness Mapping* transformation, the weaknesses represented in the *Parameterization Model* are annotated to the nodes in the coupling model. The *Weakness Mapping* sub-transformation uses the *Parameterization Model* and the coupling model after the *Completion* transformation as input and provides the coupling model as output. As first step, the *Weakness Mapping* transformation has to determine the node in the coupling model that maps to the implementation element referenced in the *Parameterization Entry*. In Figure 5.8, the statement *Cipher.getInstance("DES")* is represented

as a separate node which identifies the call of the method *Cipher.getInstance*. When the node is resolved, it has to be annotated with the instance of *Weakness* in the *Parameterization Entry*.

Non-Compliance Detection & Integration. The *Non-Compliance Detection & Integration* transformation calculates whether a weakness detected in the *Source Code* violates a security characteristic specified in the *Annotated Architecture*. This detection is performed by *search strategies* which define the relation between the architectural elements and the source code elements to detect non-compliance. This search strategy is executed for each vulnerability representing an instance of a *Weakness* violating the security characteristic of the coupling scenario. For this purpose, the search strategies use the coupling model after the *Weakness Mapping* transformation, the *Security Relations* model, the *Parameterization Model* and the *Annotated Architecture* as input and provide the parameterized *Annotated Architecture* as output. A search strategy defines, (1) how the coupling model is traversed to determine whether a security characteristic annotated to an architectural element is violated by the vulnerability found in the implementation, and (2) which elements of the coupling model must be encountered to determine non-compliance. If the elements from (2) are encountered, the search strategy inspects if a security characteristic is annotated and whether it is violated by the weakness. The search strategy is specific to an instance of *Weakness* and is executed for all vulnerabilities which represent weaknesses that violate the security characteristic in the coupling scenario

We now illustrate a search strategy for our running example. Note this is an exemplary search strategies and other search strategies are possible for the same weakness and security characteristic. In particular, this depends on the coupling metamodel applied. For our running example, we specify a search strategy for the *Weakness weak encryption* and the coupling metamodel described in Subsection 5.2.4. Figure 5.8 illustrates how this strategy is executed on the coupling model representing the system in the running example. This search strategy is similar to a taint analysis and performed as follows:

1. The search strategy starts its traversal from the nodes annotated with the instance *weak encryption*. In our running example, this is the node representing the call of *Cipher.getInstance("DES")*.
2. The edges in the coupling model representing the flow of data affected by this node are traversed. In the running example, this includes the edges representing the flow of the *cipher* to the method *encrypt* and the flow of *encUserData* to the call of the *store* method.
3. If a *visible edge* or *hidden edge* is encountered, the search strategy investigates whether a *Security Characteristic* is annotated. In our running example, this is the case for the edge representing the communication between *buy* method and the *store* method mapping to the communication link *Link1*
4. If a *Security Characteristic* is annotated, a lookup into the *Security Relations* model is performed to determine whether it is violated by the instance of *Weakness*. This is the case in our running example because the detected *visible edge* is annotated with *encryption* which is violated by *weak encryption*.

5. If the weakness violates the security characteristic, non-compliance is reported. In the running example, this is the case for the weakness *weak encryption* which violates the security characteristic *encryption*.

In the search strategies of our coupling approach, we currently do not consider the semantics of the implementation elements or architectural elements mapping to the nodes. Considering the semantics would require information about the functionality of implementation elements or architectural elements. For example, the search strategy is not aware that a node represents encryption functionality. Thus, the coupling approach cannot determine whether encryption weaknesses remain throughout the whole system for the affected data. *Cipher.getInstance("RSA")* in the *store* method in our running example is not considered *weak encryption*. As a result, the search strategy continues its traversal over the corresponding node, because it is not aware that *encUserData* now contains *userData* which is correctly re-encrypted. Information about the semantics of the nodes can be obtained by using additional information sources such as SecDFDs [338]. However, using such information sources requires performing additional analysis to connect them to the implementation as demonstrated by Peldszus et al. [256] which is out of scope of this dissertation. Consequently, to address this challenge, we do not consider the semantics of nodes in our current coupling approach but perform approximations. We discuss the limitations resulting from not considering the semantics of nodes in Section 5.3 and outline how to address these limitations as part of future work in Section 11.3.

These approximations are performed by a *Stop Criterion* for each search strategy, specifying when the traversal of the coupling model is stopped. However, this criterion affects the accuracy of the coupling by potentially introducing false positives or false negatives. An exemplary *Stop Criterion* is that coupling model is traversed until no further edge is available. In our running example, this stop criterion would result in propagating the *Weakness weak encryption* to the *visible edge* mapping to *Link2* because of the unawareness of the re-encryption in the *store* method. Consequently, the coupling approach would report non-compliance for *Link2* because it is annotated with *encryption* (*false positive result*). A *Stop Criterion* which terminates the traversal of the coupling model when a *visible edge* is encountered would address this false positive result in our running example. This stop criterion is based on the assumption that data is not communicated to another component without being re-encrypted. Consequently, non-compliance only originates from a *single* component. However, this stop criterion can result in missed non-compliance (*false negative result*) if the assumption is violated. For illustration, we modify the running example by removing the re-encryption of the weakly encrypted data in the *store* method. Instead, the *store* method now forwards the data directly to the *log* method. A search strategy that stops after the first encounter of a *visible edge*, would not detect non-compliance between the weakness and the annotation for *Link2* (*false negative result*). Please note how a weak re-encryption in the *store* method (e.g., by selecting *DES* as encryption algorithm again) would correctly result in non-compliance for *Link2* with this stop criterion. This is because the search strategy would also be applied to the node representing the selection of the cipher in the *store* method.

If non-compliance is detected by the search strategy, the *Non-Compliance Detection & Integration* transformation modifies the affected annotation in the *Annotated Architecture*. The transformation first resolves architectural element in the *Annotated Architecture* mapped to the edge of the coupling model for which the search strategy determines non-compliance. For our running example, the *Non-Compliance Integration* transformation resolves the communication link *Link1*. After the architectural element is resolved, the annotation which annotates the security characteristic of the coupling scenario has to be calculated. If the annotation specifies the values of the security characteristic (e.g., true or false), the transformation has to compare for every annotation the type of the security characteristic annotated to resolve the annotation. Then the transformation has to replace the value by the one indicating that the security characteristic is not applied. In case that the value of the security characteristic is represented by the presence or absence of an annotation, the transformation has to identify the type of annotation which represents the security characteristic and select it accordingly. With this information, the transformation can remove the respective annotation.

5.2.6. Discussion: Efforts of the Pattern-Search-Based Coupling

We now discuss efforts that arise when realizing the pattern-search-based coupling process. Similar to the discussion of effort in the specification-based coupling process (see Subsection 5.1.5), we will categorize the efforts into *one-time efforts* for creating transformations that can be *automated* and *repeated efforts* that must be performed more than once in the coupling process *manually*. We structure this discussion along the processing steps of our pattern-search-based coupling approach (illustrated in Figure 5.9).

5.2.6.1. Efforts in the Definition of Security Relations Model Step

Creating the *Security Relations* model is a *fully manual task*. However, effort for this task arises only if (1) a *Security Relations* model does not include an instance of a *Security Characteristic* to which the security characteristic in the coupling scenario can be mapped or (2) if the source code analysis reports patterns indicating weaknesses that cannot be mapped to existing instances of *Weaknesses* in the *Security Relations* model. In these cases, the coupling expert has to manually create a new *Security Relations* model or extend an existing one by adding the necessary *Security Characteristics* or *Weaknesses*.

If a coupling scenario uses architectural analyses and security characteristics which map to instances of *Security Characteristic* captured in an existing *Security Relations* model, then only the transformations have to be defined. UMLSec [154] and Access Analysis [172] both annotate security characteristics defining *encryption* by specifying the presence or absence of the annotation. Thus, both annotations map to the *Security Characteristic encrypted*. Consequently, the coupling expert has to perform a one-time effort for creating the *Security Relations* model while it is applicable for coupling scenarios investigating non-compliance for both security characteristics. The same principle applies when the pattern-search-based

source code analysis is varied. The coupling expert has to define no *Weaknesses* in the *Security Relations* model if the existing ones correspond to the weaknesses indicated by the patterns detected by the source code analysis. Analyses like PMD [260], SonarQube [313], or Snyk [310] detect the application of *DES* which maps to the *Weakness weak encryption*. When we perform couplings for these analyses, the instances of *Weakness* have to be defined only once.

5.2.6.2. Efforts in the Analyses Input Model Consistency Step

Defining the *Annotated Architecture* as input for the *Analyses Input Model Consistency* step is a fully manual task, because we comprehend it as prescriptive design of the software system. Consequently, we cannot apply, for example, reverse engineering techniques to align it with the *Source Code*.

The *Analyses Input Model Consistency* comprises three major transformations *Consistency*, *Architectural Specification Mapping* and *Completion*. The *Consistency* transformation comprises *one-time effort* to provide transformations for (1) establishing a *Source Code* which is consistent to the *Annotated Architecture* (Establishing Consistency sub-transformation) and (2) generating the coupling model capturing corresponding elements (Structural Mapping (Consistency) sub-transformation). The effort for defining the *Establishing Consistency* sub-transformation is reduced compared to the specification-based coupling approach, because only transformation rules between the structural elements have to be described. Only addressing the structural elements can reduce the effort because they target the architectural system design which can be reused in different analyses. For example, the PCM for example is used in various security analyses [112, 172, 300, 348] and is often mapped to the implementation (e.g., in [120, 167, 173, 181, 213]). In contrast, the architectural specification is specific to the individual analyses as it captures the elements necessary to analyze the respective security property. The *Structural Mapping (Consistency)* sub-transformation results in a *one-time effort* to generate the edges and nodes which map to the *Annotated Architecture* and the *Source Code*. This effort is assumed to be small because the correspondences between the *Annotated Architecture* and the *Source Code* are already defined in the *Establishing Consistency* sub-transformation.

The *Architectural Specification Mapping* transformation results in a *one-time effort* to define transformation rules that annotate the edges in coupling model with the *Security Characteristics* of the *Security Relations* model which map to the security characteristic of the coupling scenario. This transformation requires to determine the edges that map to architectural elements which imply communication annotated with the security characteristic in the coupling scenario. The transformation between edges and these architectural elements is already defined in the *Structural Mapping (Consistency)* sub-transformation. Thus, the effort for this task can be reduced by reusing the respective transformation rules. The effort can be reduced further if the prior transformation yields a correspondence model that contains the correspondences between the edges and the *Annotated Architecture*. In this case, determining the mapping between an architectural element and an edge is reduced to a lookup in the correspondence model. Thus, *one-time effort* arises to define

the transformation (1) to determine whether an annotation exists for the element in the *Annotated Architecture* that annotates the security characteristic in the coupling scenario, and (2) to map the security characteristic to the *Security Characteristic* in the *Security Relations* model. The mapping between the security characteristic of the coupling scenario and the *Security Characteristic* in the *Security Relations* model is low as the latter is built by the coupling expert so that it captures the respective security characteristic.

The *Completion* transformation includes the two sub-transformations *Source Code Completion* and *Structural Mapping (Completion)*. The *Source Code Completion* sub-transformation results in *repeated effort* for manually completing the *Source Code*. This effort arises for all parts of the *Source Code* that have to be adapted in an evolution step. In contrast, the *Structural Mapping (Completion)* sub-transformation results in *one-time effort* for creating the transformation that provides the nodes and edges for which no corresponding elements in the *Annotated Architecture* exist. We demonstrate this automation in our prototypical implementation of our coupling approach [269] where we use the framework SPOON [249] to parse Java source code to extract the nodes and edges for the coupling model.

5.2.6.3. Efforts in the Non-Compliance Analysis Step

The effort in the *Non-Compliance Analysis* is limited to executing the analyses. If the analysis only allows execution through a graphical user interface, *repeated effort* arise for this task. This effort can be considered small for the coupling expert as they only have to understand the user interface once. However, commonly pattern-search-based source code analyses provide command-line interfaces or an API to execute the analysis which allows automated execution. This automation results in a *one-time effort* to define the execution of the source code analysis. Efforts also arise if the source code analysis requires selecting the patterns to be detected. Then the coupling expert has to define for each security characteristic in the coupling scenario and source code analysis, which patterns indicate weaknesses violating the security characteristic. However, if the source code analysis can be executed automatically, the effort for selecting the patterns can also be automated. This is because the relation between patterns and a dedicated security characteristic does not change in every execution of the coupling approach.

5.2.6.4. Efforts in the Parameterization Step

The effort in the *Parameterization* step includes the definition of the *Parameterization Model Resolution* transformation and the *Integration* transformation.

The *Parameterization Model Resolution* transformation must map (1) the reported location of a pattern in the *Source Code Analysis Result* to implementation elements of the *Source Code* and (2) the pattern reported to the respective weaknesses. The analysis architect defines how the source code location where a pattern is detected is represented in the *Source Code Analysis Result*. They know how to parse the syntax and know the relation between the location in the *Source Code Analysis Result* and the implementation elements

in the *Source Code*. Consequently, we expect the effort for this task to be low for the analysis architect. Similarly, the analysis architect defines the patterns to be reported. These patterns are detected because they indicate weaknesses in the implementation which are relevant for the analysis architect. Consequently, we expect the effort for defining a transformation from the patterns represented in the *Source Code Analysis Result* into *Weaknesses* defined in the *Security Relations* model to be low. This effort arises once for every pattern in a source code analysis. However, this effort arises regardless of whether the *Source Code Analysis Result* contains the weakness associated with the pattern or not. The transformation either contains the relation between the pattern identifier or the weakness identifier. Thus, both identifier must be parsed and mapped to instances of *Weakness* in the *Security Relations* model.

For the *Integration* transformation we assume *one-time effort* for creating the *Weakness Mapping* sub-transformation, for creating the search strategies in the *Non-Compliance Detection* sub-transformation and for defining the *Non-Compliance Integration* sub-transformation. The *Weakness Mapping* transformation requires calculating the node that maps to the implementation element captured in a *Parameterization Entry*. As this relation is already established in the *Analyses Input Model Consistency* step, we see the effort for this task as low. The *Parameterization Model* and the coupling model both include instances of *Weakness* in the *Security Relations* model. Consequently, the effort of annotating the instances is low because it is reduced to creating an annotation in the coupling model with the *Weakness* in the *Parameterization Model*.

A major effort for the coupling expert is to define the *search strategies*. These strategies determine the semantic relations between the elements in the *Annotated Architecture* and the *Parameterization Model* necessary to perform the parameterization. Consequently, errors in these strategies can lead to false results of the coupled analysis due to erroneous calculation of non-compliance. We expect the coupling expert to invest major effort in creating and testing the search strategies. The number of search strategies created or the frequency that the strategies have to be adapted depends on the addressed *Security Characteristics*, detected *Weaknesses* and changes of the coupling metamodel.

Addressed Weaknesses. A search strategy is defined for a weakness and a coupling metamodel. Thus, if a source code analysis applied in a coupling scenario detects a pattern that relates to a weakness not yet captured in the *Security Relations* model, a new search strategy has to be defined.

Addressed Security Characteristics of the Coupling Scenario. If a coupling scenario addresses a security characteristic that cannot be mapped to the *Security Relations* model, a new *Security Characteristic* in this model has to be defined. Thus, all search strategies for weaknesses that violate this security characteristic have to be adapted. The effort for this adaptation depends on the realization of our pattern-search-based coupling approach. Our prototypical implementation shows a worst-case scenario where the transformation rules are encoded into the search strategy. This approach introduces the necessity for the coupling expert to invest effort to create new search strategies when the architectural analysis is exchanged.

Coupling Structure. The coupling metamodel defines how the elements in the *Annotated Architecture* and the *Source Code* are related. Consequently, using a new coupling metamodel introduces effort for the creation of new search strategies for all weaknesses and security characteristics that have to be addressed in a coupling scenario.

The integration of non-compliance requires (1) the lookup of the architectural element mapping to the edge that is affected by the non-compliance and (2) the lookup of the annotation specifying the non-compliant security characteristic for that architectural element. Both mappings are already defined in the *Analyses Input Model Consistency*. Thus, the coupling expert already knows how to calculate the mappings between the respective elements, reducing the effort for this transformation. The effort for creating this transformation can be reduced further if a correspondence model is created in the *Analyses Input Model Consistency*. With such a correspondence model, the transformation is reduced to a lookup for the corresponding element in the model.

5.3. Limitations

In this section, we discuss the limitations of the specification-based coupling approach and pattern-search-based coupling approach. Future work to address these limitations is discussed in Section 11.3. We previously identified six limitations **L1** - **L6** of the coupling design in Section 4.5. In this section, we discuss five additional limitations **L7** - **L11** that arise from the coupling approaches.

Limitation 7: Limited Generalizability Both coupling approaches focus on specific security properties rather than providing a general framework applicable to a wide range of security concerns. The specification-based coupling approach focuses on information flow security while the pattern-search-based coupling approach focuses on encryption. Focusing on specific security properties limits the generalizability of the coupling approach. However, both topics are relevant in research and industry as motivated in Section 5.1 and Section 5.2. Thus, the focus on information flow security to illustrate a coupling approach based on the semantically identical relation type and focusing on encryption to illustrate a coupling approach based on the semantically related relation type is justified.

Furthermore, although we describe the coupling approaches for these security properties, we formulate the ICs in the specification-based coupling approach and the mechanisms for detecting the semantic relations on a rather abstract level. Consequently, general applicability to other security properties is possible. However, we do not provide evidence for the applicability of the coupling approaches to other security properties; thus we do not claim generalizability.

Limitation 8: No Constructive Guidance for Transformations The ICs and reference metamodels in the specification-based coupling approach form a coherent foundation for

creating the transformations in the coupling. The same applies for the reference metamodels and the coupling model in the pattern-search-based coupling approach. However, we provide no guidance for creating the transformations in either coupling approach. Constructive guidance is important to support coupling experts in applying our coupling. This is highlighted by first investigations for the specification-based coupling approach on how to address different semantics of security characteristics of data in the coupling in the master's thesis of Lehmann [185]. This investigation shows that different semantics, specifically for the semantics of *disjunctive roles* can require transformations distinct from other semantics and a different approach on how to execute the analysis coupling. However, we have to understand the information and relations necessary for the coupling and mechanisms to perform the coupling before we can provide constructive guidance for creating the transformations. Therefore, first focusing on defining the necessary information and relations for the coupling and mechanisms to perform the coupling is justified.

Limitation 9: Manual Creation of Security Relations Model The relation between *Security Characteristics* and *Weaknesses* in the *Security Relations* model must be created manually in our pattern-search-based coupling approach. This task requires expert knowledge in the security domain. The coupling expert must be aware of all *Weaknesses* that violate a *Security Characteristic*. Thus, the manual creation of the *Security Relations* model is prone to errors regarding completeness and correctness. This limitation affects the correctness of the coupling results because incorrect or incomplete relations can lead to undetected non-compliance or false positives.

Limitation 10: Lack of Semantics in Search Strategies in the Pattern-Search-Based Coupling Approach In Subsection 5.2.5, we discuss the challenge regarding accuracy because we do not consider the semantics of the architectural elements or implementation elements mapping to the nodes in the coupling model. For example, it is unclear whether a node in the coupling metamodel represents an encryption of data. This lack of considering semantics requires approximation by the *Stop Criterion* and can lead to inaccurate results as discussed in Subsection 5.2.5.

A potential solution to this problem is to consider additional information sources such as SecDFDs [338] used by Peldszus et al. [256] to determine semantic details of implementation elements. However, using additional information sources or analyses is out of scope for our coupling approaches, which focus on coupling two analyses without the need for additional analyses. Consequently, we see the use of the *Stop Criterion* as a first step to address the lack of semantics in the search strategies, while future work can investigate how to consider semantics by additional information sources.

Limitation 11: Binary Values for Security Information in the Pattern-Search-Based Coupling Approach In Subsection 5.2.2, we assume binary values for security characteristics of the coupling scenario in the *Annotated Architecture* of the pattern-search-based coupling approach. However, approaches exist that use several values to describe the security of a

communication link. In *UMLSec* [154], for instance, the security of a communication link can be annotated with the values *LAN*, *Internet* or *Encrypted* to express different degrees of security. Thus, our assumption of binary values limits the generalizability of the pattern-search-based coupling approach. To support such approaches, the coupling expert has to consider the semantics of the values in the transformation that modifies the annotation. Thus, when encryption is violated, the transformation can modify the annotation to *Internet* to express that the communication is not encrypted but still performed over the internet where the data is prone to be read, modified or deleted by unauthorized actors. However, for the coupling it is not obvious whether the system is designed so that the communication is performed over *LAN*. Consequently, defining transformations accounting for non-binary values can prove challenging for the coupling expert because the transformation depends on contextual knowledge about the system on the model level.

5.4. Summary And Outlook

In this chapter, we presented two coupling approaches that instantiate the coupling design as our second contribution **C2** to answer ❓ **Research Question 2**, ❓ **Research Question 3** and partially ❓ **Research Question 4**. The coupling approaches address ❌ **Problem 1** and ❌ **Problem 2** by introducing concepts and mechanisms that establish relations between the (meta)models of an architectural security analysis and a static source code security analysis, enabling the detection of architectural vulnerabilities arising from a non-compliant implementation.

Each coupling approach addresses one of two major types of static source code analyses [49]: *specification-based* source code analyses and *pattern-search-based* source code analyses. Furthermore, the coupling approaches address different types of security characteristics that are specified in the architectural analysis and influenced by the implementation. The specification-based coupling approach investigates security characteristics of data influenced by information flows. Information flow security is an important property as vulnerabilities in information flows can lead to violation of the confidentiality and integrity of sensitive data [20, 63]. The pattern-search-based coupling approach investigates security characteristics specifying the need for data encryption. Data encryption is the focus of security standards and regulations (e.g., by NIST [21, 22] or the German Federal Office for Information Security [87]) because it is a widely used security mechanism to protect sensitive data from unauthorized access [318].

Differences in the information contained in the input and output models of the addressed analyses, as well as in the considered security characteristics, result in different semantic relation types addressed by each coupling approach. The specification-based coupling approach exploits the semantically identical relation type, and the pattern-search-based coupling approach addresses the semantically related relation type. This difference influences the focus of the coupling approaches when establishing the coupling.

The coupling approaches are structured similarly by detailing the reference metamodels, defining non-compliance, specializing the coupling process, and discussing the effort required to apply each approach. However, the focus of the coupling approaches differs based on the type of semantic relations they address. We summarize the coupling approaches as follows.

Specification-Based Coupling Approach The specification-based coupling approach couples an architectural analysis with a specification-based source code information flow analysis to detect architectural vulnerabilities that arise from information flows in the implementation and violate the architectural specification (Section 5.1). In this coupling approach, the architectural analysis detects architectural vulnerabilities using security characteristics of data (e.g., specified for parameters of services) that are influenced by information flows. The specification-based source code information flow analysis detects information flows in the implementation that violate security characteristics of data (e.g., specified for parameters of methods, or fields of classes).

We first introduce reference metamodels to define the information required for the coupling. In this coupling approach, both analyses use system representations comprising annotations of security characteristics to system elements (e.g., parameters of services in the architectural model and parameters of methods in the source code model). The reference metamodel for the source code analysis captures counter-examples representing information flows by their sources and sinks as well as the security characteristic resulting in the violation. Based on this information, the specification-based coupling approach focuses on non-compliance that arises from information flows in the implementation and results in values of the security characteristics of data that violate the specifications of the security characteristics of data in the architectural model.

The specification-based coupling approach focuses on the semantically identical relation between the information in the input and output (meta)models of the analyses in the coupling. Once the semantically identical relations are determined by the coupling expert, the mechanisms to operationalize the calculation of the relations between the syntactic elements of the models of the analyses for the coupling are less complex. The key challenge is that information with the same semantics in the (meta)models of the analyses must be determined by the coupling expert.

We support this task by formulating necessary but not sufficient conditions (called Integration Conditions (ICs)) between the (meta)models of the analyses. These conditions define relations between the (meta)models of the analyses that enable detection of non-compliance and the parameterization of the *Annotated Architecture* with the values of the security characteristics of the coupling scenario resulting from the implementation. The ICs enforce that (1) the values of the security characteristics resulting from the implementation can be related to the values of the security characteristics of the coupling scenario specified in the architectural model, and (2) the annotations of the security characteristics in the architectural model can be retrieved from the information available in the *Source Code Analysis Result*. To abstract from specific analyses, we define the ICs based on the

presented reference metamodels. The ICs are specified on the metamodel and model level (due to model-level semantics introduced in Subsection 4.3.3) and classified into *Correspondence Conditions* and *Constraint Conditions*. Correspondence conditions describe required correspondences *between* the elements of two (meta)models of the analyses. Constraint conditions describe required references from elements in a single (meta)model to elements that are affected by correspondence conditions of the same metamodel.

We then specialize the coupling process to address the information and relations relevant for the coupling of architectural analyses and specification-based source code information flow analyses. In this process, the *Consistency* transformation has to ensure consistency between the system descriptions and the specification. If consistency between the annotation of the values of security characteristics of the coupling scenario in the *Annotated Architecture* and the annotations of the security characteristics in the *Annotated Source Code* is established, the reports of the source code analysis indicate non-compliance with the architectural specification. This relation is used to calculate the values of the security characteristics resulting from the implementation and to integrate these values into the *Annotated Architecture*. Throughout the coupling process, we discuss the tasks of the transformations in relation to the ICs so that the necessary relations between the (meta)models of the analyses are established.

Pattern-Search-Based Coupling Approach The pattern-search-based coupling approach couples an architectural analysis with a pattern-search-based source code analysis to identify architectural vulnerabilities that arise from encryption-related vulnerabilities in the implementation (Section 5.2). In this coupling approach, the architectural analysis detects architectural vulnerabilities based on security characteristics that imply the need for data encryption specified for architectural elements describing communication. The source code analysis identifies patterns in the implementation that indicate vulnerabilities that can invalidate data encryption.

We first specify the information required for the coupling in the form of reference metamodels. Here, the reference metamodel for the *Annotated Architecture* captures the security characteristics of the coupling scenario and their annotations to architectural elements describing communication. In contrast to the specification-based coupling approach, the *Source Code* only represents the system without annotations of security characteristics. The *Source Code Analysis Result* captures locations of patterns found in the implementation that indicate vulnerabilities. Based on this information, the pattern-search-based coupling approach focuses on non-compliance that arises from vulnerabilities in the implementation and violates the need for data encryption specified in the architectural model. In particular, the non-compliance can only be determined for *binary* value ranges of the security characteristics of the coupling scenario (e.g., *encrypted* vs. *not encrypted*), because the patterns indicating vulnerabilities in the implementation only allow to determine whether a vulnerability is present or not.

The pattern-search-based coupling approach focuses on the semantically related relation type between the elements in the (meta)models of the analyses in the coupling. The

semantically related relation type requires techniques to identify related elements in the models of the analyses during the execution of the coupling. We address this task by defining two metamodels that allow us to calculate the relations between the models of the analyses to be coupled: the *Security Relations* metamodel and the *Coupling Metamodel*. The *Security Relations* metamodel allows us to capture the violation of *Security Characteristics* by *Weakness*. The *Weakness* and *Security Characteristic* are defined as abstractions of specific security characteristics in the architectural model and vulnerabilities to be found by source code analyses. The *Coupling Metamodel* captures all information necessary to calculate the semantic relations between the input and output models of the analyses in the coupling approach. We illustrate the coupling approach using a call-graph-based coupling model with data flow information. This coupling model contains nodes, edges, and data communicated over the edges. The nodes represent services in the architectural model and methods in the implementation. The edges represent communication between services or methods. To capture the security-relevant information in the coupling, the edges are annotated with the metaclass *Security Characteristic* from the *Security Relations* metamodel and the nodes are annotated with the metaclass *Weakness*.

We specialize the coupling process to focus on creating the coupling model and applying *Search Strategies*. We introduce the *Security Relations Model Definition* process step in which the *Security Relations* metamodel is instantiated for the coupling scenario by defining the relevant security characteristics and weaknesses for the coupling scenario. The coupling process creates the coupling model, by (1) mapping services and methods to nodes, (2) mapping architectural elements representing communication and communication resulting from the implementation to edges, (3) mapping annotations of the security characteristics in the *Annotated Architecture* to references from nodes to security characteristics of the *Security Relations* model, and (4) mapping patterns detected in the implementation to references from nodes to weaknesses of the *Security Relations* model. With the established coupling model, search strategies are applied to determine non-compliance and to perform the parameterization. Each search strategy defines for a weakness how the coupling model must be traversed to determine whether the weakness violates the security characteristics of the coupling scenario specified in the *Annotated Architecture*. We identify the need for defining a *Stop Criterion* in the search strategies to address the lack of semantics of implementation elements and the resulting inaccuracy of the search strategies.

Limitations In addition to the previously identified limitations of the coupling design, the coupling approaches introduce five further limitations. Notably, both coupling approaches are specific to certain types of static source code analyses and security characteristics, which limits their generalizability. Furthermore, we provide no constructive guidance for deriving transformations in either coupling approach, even though such guidance is important to support coupling experts in applying the coupling. The pattern-search-based coupling approach has three additional limitations: (1) the manual creation of the *Security Relations* model is prone to errors regarding completeness and correctness, (2) the lack of semantics in the search strategies can lead to inaccurate results, and (3) the assumption

of binary values for security characteristics limits the generalizability of the coupling approach.

Outlook The presented coupling approaches illustrate how the coupling design can be instantiated for specific types of static source code analyses and security characteristics under investigation. However, applying the coupling approaches requires the selection of source code analyses for the coupling with an architectural analysis. These source code analyses must be capable of detecting non-compliance regarding the security characteristic of the coupling scenario. This selection requires determining relations between the vulnerabilities calculated by the security analyses and the security characteristics described in the *Annotated Architecture*. Here, the challenge arises that the relation between results of source code analyses and security properties — commonly described in security DSLs [174] that are used to describe the *Annotated Architecture*— is not yet systematically captured [367]. To address this challenge, we present our third contribution **C3** in the next chapter of this thesis (Chapter 6), which allows determining relations between source code analyses and security DSLs. These relations imply that the source code analysis is capable of detecting vulnerability that invalidate the security properties specified by the security DSL.

Afterwards, we present an evaluation of the coupling approaches in Chapter 8 to investigate their applicability and effectiveness in detecting architectural vulnerabilities arising from a non-compliant implementation.

6. Relating Security Domain Specific Languages and Static Source Code Security Analyses

> **Literature:** This chapter is based on the following (co-) authored publications: "*Can I Check What I Designed? Mapping Security Design DSLs to Code Analyzers*" [254] (major revision under review in the journal *ACM Transactions on Software Engineering and Methodology* at the date of submission of the dissertation)

This chapter presents our third contribution **C3**, which addresses **Research Question 4** by providing a systematic approach to relate security DSLs and static source code security analyses. Our analysis coupling aims to detect architectural vulnerabilities that arise from values of security characteristics resulting from the implementation (see Chapter 4). For this purpose, source code analyses must be applied that detect vulnerabilities in the implementation that indicate non-compliance regarding the values of security characteristics specified in the *Annotated Architecture* of an architectural analysis. The coupling approaches (presented in Chapter 5) define the information necessary to couple *specific* analyses. However, they do not support the *selection* of source code analyses that are capable of detecting such non-compliance. However, this selection is challenging because the relationship between vulnerabilities found by static source code analyses and security properties of the system — commonly described in security DSLs [174] and used to describe the input of architectural analyses — is not systematically captured [367]. As we will show later in our evaluation in Chapter 9, the lack of such a systematic approach is critical because even security experts do not always fully understand how source code vulnerabilities detected by static source code analyses affect the security properties of the system expressed with a security DSL.

To support the coupling expert in selecting a source code analysis appropriate for detecting non-compliance between the *Annotated Architecture* and the implementation, we propose *SecLan*. *SecLan* provides the *SecLan* conceptual model, which contains concepts and relations between these concepts extracted from 66 security DSLs and 408 security checks from 33 static security source code analyzers. Using these concepts and relations, *SecLan* can determine relationships indicating that a source code analysis detects weaknesses that potentially violate security properties specified in security DSLs. Note that we distinguish in *SecLan* between the analysis tool and the investigations it can perform *for the implementation level* to enable a more thorough investigation of their concepts.

Consequently, in the remainder of Chapter 6, we refer to source code analysis tools as *analyzers* and to the investigations it can perform as *security checks* (in accordance with the terminology introduced in Section 2.4).

SecLan connects to our analysis coupling because security DSLs are commonly used to describe input models of architectural analyses (e.g., in [12, 150, 154, 172, 192, 338, 348]). Thus, relationships calculated by *SecLan* indicate that the source code analysis can be used to check for non-compliance regarding the input models of architectural analyses which use security DSLs. However, *SecLan* is not limited to architectural analyses and can be applied to other abstractions of the software system such as DFDs or business process models.

The remainder of this chapter is structured as follows: First, we present our research method applied to create *SecLan* in Section 6.1. This method is used to extract the *SecLan* conceptual model which is presented in Section 6.2. We then describe our approach to relate security DSLs to source code analyses based on the *SecLan* conceptual model in Section 6.3. Next, we discuss limitations of *SecLan* in Section 6.4. Finally, we summarize the contribution of *SecLan* in Section 6.5.

6.1. Research Method

We start by introducing the method we applied to synthesize the *SecLan* conceptual model. This method is illustrated in Figure 6.1. To synthesize the *SecLan* conceptual model, we conducted a systematic review [266] of state-of-the-art design-time security DSLs and static source code security analyses. We first present the data sources we used to identify relevant security DSLs and static source code security analyses in Subsection 6.1.1. Based on the information obtained by the review, we performed a systematic process to synthesize the *SecLan* conceptual model by extracting common concepts and relationships between these concepts. This synthesis process is described in Subsection 6.1.2.

6.1.1. Data Sources

For the synthesis of the *SecLan* conceptual model, we searched for survey papers to identify relevant security DSLs and source code analyzers. In addition, we complemented these sources by expert knowledge about existing security DSLs and source code analyzers. All selected security DSLs and analyzers are listed in Table 6.1 and Table 6.2 respectively. Our replication package [270] contains the list of all considered survey papers as well as the identified security DSLs and source code analyzers.

6.1.1.1. Security DSLs

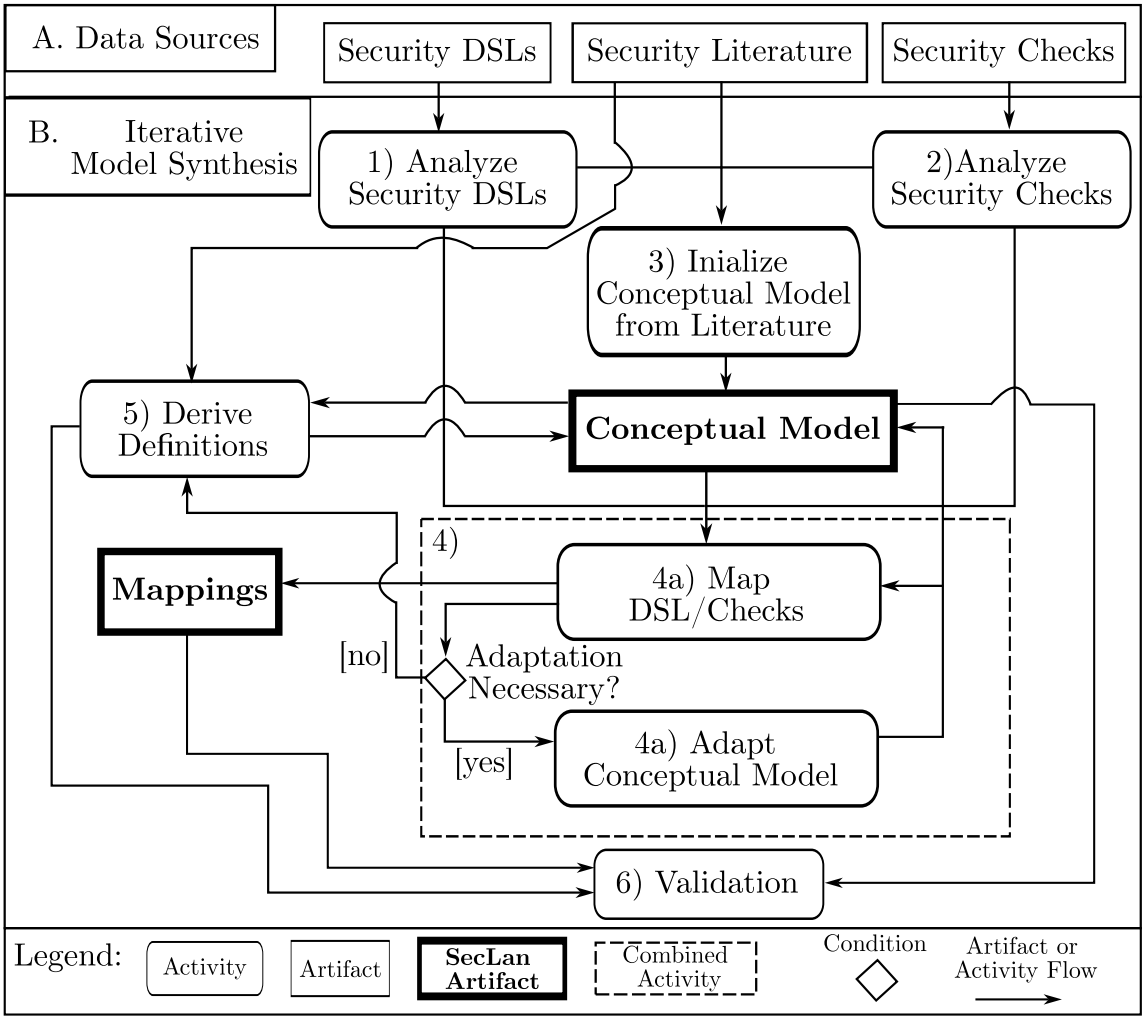


Figure 6.1.: Visualization of the process to synthesize and validate the *SecLan* conceptual model.

Table 6.1.: Overview of the Security DSLs Analyzed. The table shows the *Name* of the DSL, a *Description*, and references to relevant publications (*Ref.*).

	Name	Description	Ref.
1	AC-PIM	Platform Independent Model for access control, separating policies, access decisions, and enforcement.	[43]
2	Access Analysis	Extends the Palladio Component Model with confidentiality-related security information.	[172]
3	ADM-RBAC	An extension to ADM that allows to model role-based access control.	[64]
4	Ahn et al.	The metamodel of a role-based access control model.	[5]
5	AMF	Assurance Management Framework to ensure security models are fully realized and employed.	[6]
6	AoASM	An UML extension for aspect-oriented attack scenario models and intrusion detection aspects.	[369]
7	AORDD	An ontology that describes the underlying concepts of the AORDD framework.	[94]

Continued on next page

Table 6.1 – continued from previous page

	Name	Description	Ref.
8	ASM	UML profile for aspectual scenario modeling focusing on non-functional requirements.	[290]
9	Break-Glass RBAC	Integrates break-glass policies for emergencies into business process models.	[293]
10	Busch et al.	An extension of the UML-based Web Engineering (UWE) language to model security concepts.	[44]
11	CARDS	A language for specifying data flow constraints and architecture-level assumptions.	[150]
12	CCMSec	An extension of the CORBA Component Model (CCM) with security concepts.	[276]
13	DiaAspect/DiaSpec	A light-weight Architecture Description Language.	[143]
14	DRBD-AFT	A combined model of dynamic reliability block diagrams (DRBD) and attack-fault trees (AFT).	[179, 180]
15	FDAF Metamodel	A UML extension to weave the security aspect into a UML architecture design.	[58]
16	Gomaa et al.	Secure software architectures based on secure connectors for inter-component communication.	[303]
17	Hoisl et al.	UML extension for secure object flow at the PIM level.	[132]
18	Horcas et al.	Annotations used for the dynamic monitoring of security policies.	[134]
19	Kim et al.	Describes user assignment to roles and permissions of roles to access objects and operations.	[163]
20	Koch et al.	A language for role-permission assignments.	[168]
21	Menzel et al.	Enables modeling of service-oriented architecture	[205]
22	Morin et al.	Specifies an access control policy and connections to the software architecture.	[216]
23	Mouelhi et al.	Describes elements for specifying, deploying and testing of access control policies	[217]
24	Mouheb et al.	The Meta-Model for Specifying Security Hardening Plans	[218]
25	Nakamura et al.	Stereotypes to address business-level security intents.	[227]
26	Nguyen et al.	Allows to specify access control and delegation policies.	[232]
27	Oladimeji et al.	A lightweight extension to the UML with authorization and obligation policies.	[237]
28	OrBAC EMF	A realization of the OrBAC model in EMF.	[156]
29	OSL-ISDL	Adds references to the Interaction System Design Language to the Obligation Specification Language.	[229]
30	Pavlich-Mariscal et al.	A UML profile for specifying access control diagrams.	[247]
31	PbSD	A pattern-based method to describe role-based access control for objects and actors in the system.	[2]
32	Pinto et al.	Models and links security concerns concerns to a system architecture model.	[133, 259]
33	RBAC DCM	A hierarchical RBAC design class model to describe RBAC classes and operations.	[362]
34	RBML	A UML-based Role-Based Metamodeling Language.	[162]
35	SAL	A Security Analysis Language to analyze flows of data objects through distributed systems.	[75]
36	SAM Security Extension	Defines in the SAM which information is sensitive to which action/-task.	[362]
37	Satoh et al.	A model including platform information required for creating security policies.	[292]

Continued on next page

Table 6.1 – continued from previous page

	Name	Description	Ref.
38	SecBPMN	Security annotations to represent security aspects of a business process.	[288]
39	SecDSVL	Visual modeling of security details in enterprise systems security management processes.	[11]
40	SECDW	A UML profile for specifying confidentiality constraints in conceptual multidimensional modeling.	[89]
41	SecDFD	Annotations to specify data flow contracts on data flow diagrams.	[338]
42	SecML	A requirements metamodel extended with security concepts for defining security requirements.	[289]
43	SECTET-PL	A policy language influenced by Object Constraint Language and interpreted on UML models.	[8]
44	SecureMDD	Modeling of a system, its functional and security requirements, and attackers.	[212]
45	Secure Tropos	A goal-based requirements language for expressing security concerns.	[219]
46	Secure xADL	Extends xADL with Constructs for describing security characteristics.	[274]
47	SecureSOA	A domain-specific language to specify security configuration patterns.	[206]
48	SecureUML	UML-based language for the model-driven development of secure, distributed systems.	[192]
49	Security@Runtime	The metamodel for the Security@Runtime approach.	[79]
50	Security Primitives Lib.	A library of security building blocks for SPACE.	[107, 108]
51	Security Model	Allows modeling of security dysfunctional behavior.	[38]
52	Secure SaaS	Integrates security patterns into a more comprehensive pattern for handling more threats.	[215]
53	Seifermann et al.	Annotations to specify data flow-related security properties in the Palladio Component Model.	[300]
54	Sensitivities	Extends mechatronic UML with sensitivities of data received or sent from ports in components.	[96, 97]
55	Shin and Gomaa	Allows to describe security-relevant elements while separating system and security concerns.	[101]
56	SSD RBAC	A metamodel for Static Separation of Duty (SSD) and Role-Based Access Control (RBAC).	[267]
57	SysML-Sec	A security extension to SysML.	[17]
58	UMLS	Extends the UML with annotations to address confidentiality throughout the development process	[124]
59	UMLsec	UML profile defining security annotations for dependencies, information flow, and data processing.	[154]
60	VBAC-PIM	The Meta Object Facility model for the View-based Access Control Model.	[91]
61	Visual RBAC	A visual language to specify access and security policies according to role-based access control	[99]
62	Wada et al.	An UML profile that provides key model elements to specify service-oriented applications.	[346]
63	Walter et al.	An ADL extension for modeling access control policies, attacker capabilities, and system vulnerabilities.	[348]
64	Wolter et al.	A security policy model based on a simplistic view of a service-oriented architecture.	[355]

Continued on next page

Table 6.1 – continued from previous page

	Name	Description	Ref.
65	Xiao et al.	A metamodel for building adaptive and secure multi-agent systems with RBAC.	[357]
66	Xu et al.	The metamodel for aspect-oriented modeling of securanalyzed ity threats.	[358]

We identified four surveys [26, 202, 231, 340] on model-driven security engineering that mention 105 papers, of which 57 provide security DSLs. Some DSLs are mentioned in multiple surveys, so we excluded duplicates where necessary. We considered 37 security DSLs from the survey of Nguyen et al. [231]. The security DSLs listed by van den Berghe et al. [26] and Uzunov et al. [340] had a large overlap with the survey of Nguyen et al. [231]. Thus, these surveys only list 9 and 8 additional security DSLs, respectively. Mashkoor et al. [202] contributed only three security DSLs relevant for us, because the other reviewed papers included in this survey discuss model-driven security more generally. In total, 44 papers were referenced in the surveys that did not contain a security DSL, 31 of which were referenced by Mashkoor et al. [202]. Four additional references may contain security DSLs, but we did not have access to the corresponding publications or preprints. We added seven DSLs based on our expertise, as some were recently proposed. Furthermore, we added two DSLs which have been suggested by experts for the security DSLs we found in the literature. This resulted in a total of 66 design-time security DSLs for our study (see Table 6.1).

6.1.1.2. Security Analyzers

Table 6.2.: Overview of the security analyzers investigated with their *Name*, a *Description*, the number of checks they provide (*#Checks*), the categories (see Subsection 2.4.2) that the provided checks cover (*Category*), and references to relevant publications (*Ref.*). Values of *Category* are: 1) API – Security API Usage; 2) IF – Information Flow; 3) D – Dependency; 4) VCP – Vulnerable Coding Practices; 5) P – Permissions.

	Name	Description	#Checks	Category	Ref.
1	Amandroid	Inter-component data flow analysis framework for security vetting of Android apps.	5	API, IF	[353]
2	Bandit	Comprises checks to find common security issues in Python code.	40	API, VCP, P	[263]
3	BinSight	Static program analysis to attribute API misuse to its source.	1	API	[223]
4	CDRep	Repairs cryptographic misuse defects in Android apps.	1	API	[195]
5	Clang Static Analyzer	Security-related checkers of the Clang Static Analyzer.	15	VC	[326]
6	CMA	Crypto misuse analyzer based on analyzing runtime information.	1	API	[305]

Continued on next page

Table 6.2 – continued from previous page

	Name	Description	#Checks	Category	Ref.
7	CodeQL	A static analysis framework for discovering vulnerabilities across a codebase.	196	API, VCP, IF, P	[100, 214]
8	CogniCrypt	CogniCrypt is an API analysis framework for Java.	1	API	[52, 176]
9	CppCheck	Focuses on detecting undefined behavior and dangerous coding constructs.	2	VCP	[201]
10	CryptoChecker	Elicit security rules by clustering commonalities among semantic code changes	1	API	[243]
11	CryptoGuard	Scalable detection of cryptographic API misuse vulnerabilities in Java projects.	1	API	[265]
12	CryptoLint	Analysis of crypto-related information flows.	1	API	[78]
13	CryptoTutor	Detection of cryptographic misuses and suggestion of possible repairs.	1	API	[308]
14	Dr. Checker	A bug-finding tool for Linux kernel drivers.	2	VCP	[197]
15	ErrorProne	Static analysis tool for Java that catches common programming mistakes.	2	API, VCP	[102]
16	Flawfinder	Lexical analysis for vulnerabilities based on function names.	49	VCP	[354]
17	FlowDroid	Context-, flow-, field-, object-sensitive and lifecycle-aware taint analysis for Android.	1	IF	[19]
32	Frama-C	Analzer for analyzing C-Code extended by various security related Plug-Ins	15	IF, SCP	[68]
18	Hotfixer	Crypto API misuse hotfixing at Java application runtime.	1	API	[230]
19	Infer	A static analyzer provided by Facebook to detect bugs in Java and C/C++ code	2	VCP	[45]
20	JOANA	Java analyzer for information flow control-based security leaks.	1	IF	[309]
21	MalloDroid	Detects potential vulnerabilities against MITM attacks.	1	API	[84]
22	NPDHunter	Binary analysis for Null Pointer Dereference vulnerabilities.	1	VCP	[148]
23	OWASP Dependency Check	Software composition analysis tool for detecting vulnerable dependencies.	1	D	[240]
24	PeX	Static permission check error detector for Linux.	1	P	[365]
25	PMD	Rule subset that flags potential security flaws.	17	API, VCP, P	[260]

Continued on next page

Table 6.2 – continued from previous page

	Name	Description	#Checks	Category	Ref.
26	Pyre/Pysa	A tool for performing taint analysis on Python code.	1	IF	[208]
27	Rafnsson et al. ESLint rules	ESLint rules for certain classes of cross-site scripting, SQL injection, and misconfiguration vulnerabilities in JavaScript.	3	API, IF	[264]
28	Snyk Open Source	Vulnerability detection based on known security issues found in open-source libraries.	1	D	[310]
29	SonarQube	Rule subset classified according to the CWE, OWASP Top 10, or SANS Top 25.	57	API, VCP, P	[313]
30	Tang et al.	Detects over-claims of permissions in Android apps.	1	P	[325]
31	Vulvet	Multiple program analysis techniques for detection of vulnerabilities in Android apps.	1	API, VCP	[93]
32	Wang et al.	Static and dynamic analysis to detect permission gaps in Android applications.	1	P	[349]
33	Xu et al.	Analyzes crypto API usages in Android applications based on probabilistic models.	1	P	[361]

For source code security analyzers, we identified three surveys [191, 225, 366]. These surveys mention 72 source code analyzers. The 72 source code analyzers included duplicates, and analyzers that do not provide security-related checks. We excluded these analyzers accordingly. We classified all security checks by the vulnerabilities they detect and classified the analyzers into whether they are *single-purpose* or *multi-purpose analyzers*. These classifications are described in detail in Subsubsection 2.4.2.1.

From the 46 analyzers listed by Nachtigall et al. [225], only 18 provide security checks. Liu et al. [191] discuss 6 analyzers from which 5 are also listed by Nachtigall et al. [225]. Zhang et al. [366] discuss 20 cryptographic API misuse analyzers for Java, where 15 analyzers solely focus on *API Usage*, while the 5 remaining ones are *multi-purpose* analyzers, such as *SonarQube* [313]. 48 analyzers remained after removing duplicates and those that do not provide security-related checks. We classified 25 of these 48 analyzers as *single-purpose analyzers* and 23 as *multi-purpose analyzers* (see Subsection 2.4.2 for definitions).

For our study, we decided to select all identified single-purpose analyzers for an in-depth investigation. However, the 23 multi-purpose analyzers overlap substantially in the security checks they provide. For example, Find Security Bugs [18], *SonarQube* [313] and *CodeQL* [100, 214] provide similar security checks detecting usage of insecure cryptographic algorithms (*Security API Usage*) or printing the stack trace to the command line (*Vulnerable Coding Practice*). Consequently, increasing the number of selected multi-purpose analyzers does not increase the number of different security checks at some point. Therefore, we selected a subset of commercial and free-of-use multi-purpose analyzers mentioned in

surveys for different programming languages. This selection resulted in *SonarQube* [313], *PMD* [260], *Bandit* [263], *FlawFinder* [354] (contained in multiple surveys) and the *Clang Static Analyzer* [326]. We complemented them with *CodeQL* [100, 214] and *Frama-C* [68], among others, because they support *Information Flow* analysis not supported by the other multi-purpose analyzers. Since the surveys list only a few single-purpose analyzers for some categories, we added additional representatives identified by searching single-purpose analyzers in underrepresented categories. As *Information Flow* analyzers, we added *FlowDroid* [19], *JOANA* [98], and *KeY* using a specification of Greiner et al. [105]. We added *NPDHunter* [148] for *Vulnerable Coding Practices*. As *Permission* analyzers, we added *PeX* [365], which focuses on missing or problematic permission checks, and Tang et al. [325] and Wang et al. [349], which detect claiming too many permissions.

We require analyzers to provide security checks for *specific* security-related properties as a reasonable basis to establish a relation to security DSLs. Frameworks like Coccinelle [184], Soot [343] or KeY [7] enable analysis of security-related properties if appropriately extended (e.g., by providing a certain specification). For example, FlowDroid [19] and the approach Greiner et al. [105] analyze information flows. These analyzers are based on Soot [343] or KeY [7], respectively. However, the frameworks themselves (e.g., Soot and KeY) cannot be related to a specific security-related property. Without this specificity, all security DSLs could be related to any of these analyzers without further information gained to support developers on *which* property of the DSL can be analyzed. Consequently, we excluded general-purpose analysis frameworks or verification frameworks that can be extended to perform security-related checks but do not provide them themselves. We also excluded general approaches like *fuzzing* if no specific purpose is described. In contrast, we included either the framework itself if *extensions* are closely tied to it or the security-related extension directly. For example, we include the framework Frama-C [68], because it specifically lists the Plugins *EVA* and *SECUREFLOW*, which analyze the implementation for vulnerable coding practices and information flow. In contrast, we include FlowDroid [19] and the approach of Greiner et al. [105] rather than the respective frameworks they extend (i.e., Soot or KeY).

In total, we consider 33 analyzers that provide 408 security checks (see Table 6.2). The selected analyzers provide checks for different programming languages. The most frequent languages are Java (224 checks), C/C++ (138 checks), and Python (42 checks). Other languages appear only sporadically represented. Still, the most popular programming languages according to the TIOBE index [144] are well covered.

6.1.2. Model Synthesis

With the DSLs and source code analyzers identified, we aimed to extract common concepts and relations between these concepts to synthesize the *SecLan* conceptual model. We performed this synthesis by iteratively investigating the 66 security DSLs and 33 analyzers with a total of 408 security checks. In each iteration, we extracted new common concepts, extended concepts already found or generalized them if possible. To this end, we studied the selected security DSLs and analyzers based on their documentation, their descriptions in

scientific papers as well as the input and output models used by the analyzers. Specifically, Figure 6.1 shows the six steps we performed, which we explain in the following:

1. **Analyze DSLs.** We analyzed security DSLs and extracted relevant information. This information included their purpose, the considered security objectives and elements of the underlying modeling language. We started this analysis with a list of relevant properties of the security DSLs, and possible value ranges, known from surveys [26, 202, 231, 340]. We then extracted information about the security DSLs according to these properties. Finally, we extended this list with further properties identified during the analysis of the security DSLs.

2. **Analyze Checks.** We inspected the source code security checks iteratively. We extracted properties that can be used to define each check, such as the weaknesses that can be detected, or the implementation elements that are inspected. In this step, we started with the list of properties created in the step *Analyze DSLs* and refined it to include security analyzers. Additionally, we compared the properties of the analyzers with those of the security DSLs to identify properties that are common to security DSLs *and* implementation-level analyzers.

3. **Initialize *SecLan* conceptual model.** After the steps *Analyze DSLs* and *Analyze Checks*, we determined overlapping properties of security DSLs and implementation-level analyzers, or which properties were closely related. We then conceptualized these properties based on existing literature (see Section 10.2 and Section 10.4) and our observations from inspecting the security DSLs and checks. Next, we initialized a preliminary version of the *SecLan* conceptual model, focusing primarily on relevant security concerns and a common system representation.

4. **Iterative Synthesis: Map DSLs/Checks and Adapt *SecLan* conceptual model.** We then iteratively classified the security DSLs and analyzers with respect to the preliminary *SecLan* conceptual model by performing two steps. First, for every security DSL or analyzer, we checked whether the *SecLan* conceptual model could be applied to describe them, starting with the security DSLs (4a in Figure 6.1). We mapped the elements of the security DSLs and analyzers to the concepts of the *SecLan* conceptual model. Each classification has been discussed with collaborators experienced in security DSLs and static source code analyses. In case of new or mismatching concepts, possible adaptations of the *SecLan* conceptual model were discussed among the collaborators until full agreement was reached. We then updated the *SecLan* conceptual model and the classification of previously mapped security DSLs. This update potentially led to repeating the *Iterative Synthesis* step until no further adaptations were necessary (4b in Figure 6.1). We applied the same procedure to the security analyzers. This way, we ensured that the *SecLan* conceptual model is applicable to all security DSLs and analyzers studied. During this process, we obtained instances of the final *SecLan* conceptual model for each security DSL and analyzer we studied (represented as the *SecLan* artifact *Mapping* in Figure 6.1).

5. **Derive Definitions And Relationships.** After the *SecLan* conceptual model reached a state where no adaptation was necessary, we derived concrete defini-

tions of the individual elements. We identified these definitions by systematizing the justifications behind the classifications and adaptations of the previous four steps and relating them to further literature. For each element in the *SecLan* conceptual model, we discussed possible relationships between these elements based on the definitions with the same collaborators as in the previous steps. This discussion was based on further literature and the data collected while studying the security DSLs and analyzers.

6. **Validation of *SecLan* conceptual model.** After synthesizing the *SecLan* conceptual model, we validated the concepts and relationships between the concepts with experts in the domains of security DSLs and security checks. This validation is presented in part III (*Validation*) of this thesis in Chapter 9. In this validation, we investigate whether the *SecLan* conceptual model accurately captures the respective domains and whether we correctly represent the security DSLs and security checks we studied.

Applying this process resulted in the final version of the *SecLan* conceptual model, which we present in Section 6.2.

6.2. The SecLan Conceptual Model

We capture the common concepts and relations between these concepts in the *SecLan* conceptual model which we synthesized from the investigated 66 security DSLs and 33 analyzers with a total of 408 security checks.

The *SecLan* conceptual model consists of four sub-models illustrated in Figure 6.2.

Security DSL Description. The *Security DSL Description* (center top) captures common concepts in security DSLs.

Security Analyzer Description. The *Security Analyzer Description* (center bottom) captures security checks offered by static source code analyzers.

Security Model. The *Security Model* (left-hand side) captures security concepts used to establish a relation between the *Security Analyzer Description* and the *Security DSL Description*.

System Model. The *System Model* (right-hand side) captures security-relevant system elements used to establish a relation between *Security Analyzer Description* and *Security DSL Description*.

6.2.1. Security Model

We identified three common security concepts of security DSLs and analyzers that need to be considered: *Security Objectives*, *Threats*, and *Weaknesses*. These concepts provide a fundamental view on software security. Countermeasures were also analyzed in many security checks. These security checks primarily verify the secure use of cryptographic

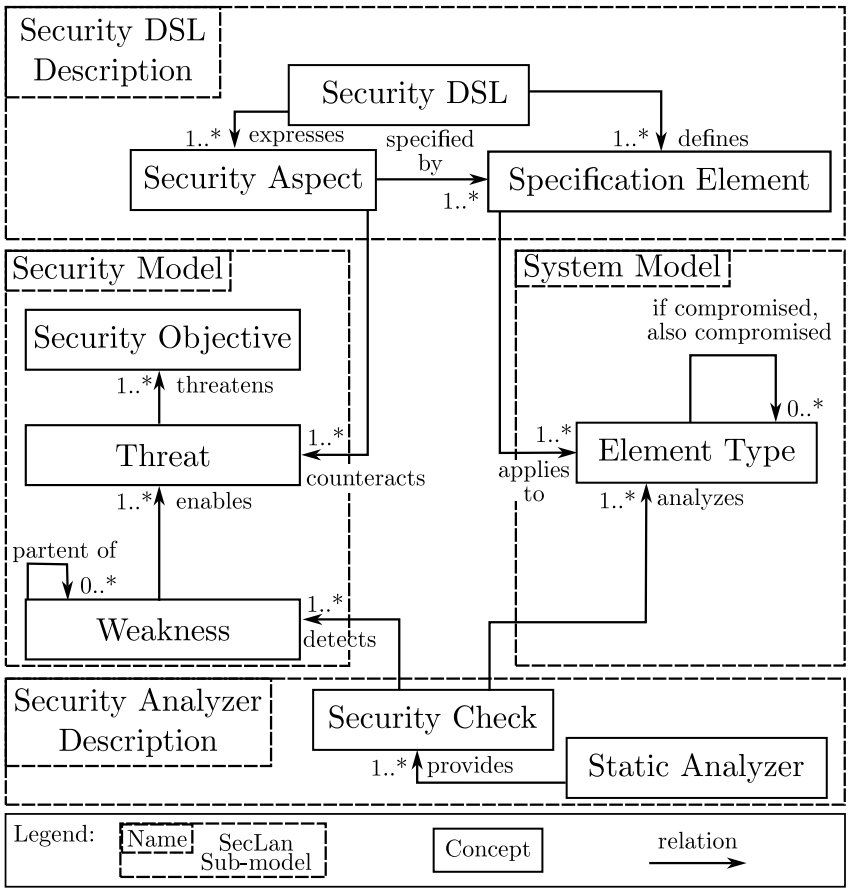


Figure 6.2.: *SecLan* conceptual model capturing the common concepts and relations between these concepts related to security DSLs and security checks.

APIs, such as CogniCrypt [176]. However, many of the examined security DSLs and security checks do not have such an explicit relationship. Consequently, we did not include countermeasures as a separate element in the *SecLan* conceptual model. Figure 6.3 shows an instance of the *Security Model*.

Security Objective All security checks and security DSLs refer to high-level security objectives. In line with the literature [299, 322, 338], we describe *Security Objectives* (called security property in ISO 27002 [140]) as high-level goals that must be achieved to ensure the security of a system. Examples of security objectives are confidentiality, integrity, and availability; also known as the CIA triad. These objectives are also captured by different institutes such as the NIST [31], in the ISO/IEC 27000 standards series [139] or in catalogs such as the CWE [327].

Besides confidentiality, integrity, and availability, we often found authorization, authentication, or access control as addressed security objectives. The first two objectives are already covered by the objectives confidentiality and integrity. These terms are also used to describe concrete security mechanisms with the goal of preventing malicious modifications, information disclosure, or ensuring the authenticity of an actor. Thus, we decided to

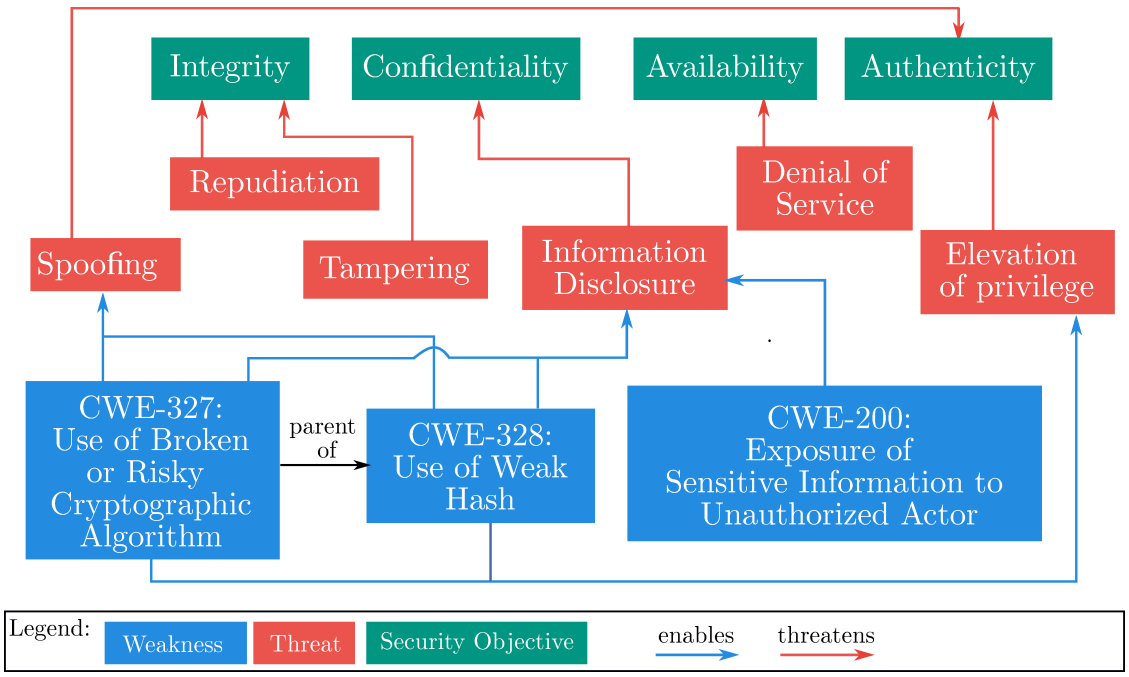


Figure 6.3.: Example for an instance of the security model based on the CIA, STRIDE, and exemplary security weaknesses from the CWE.

add authenticity as a fourth fundamental security objective in our instance of the security model in Figure 6.3. However, further security objectives such as *privacy* proposed in the LINDDUN framework [278] can also be included in the *SecLan* conceptual model.

Threat Security objectives are threatened by *Threats*. The NIST [153] defines a threat as a circumstance or event that can adversely affect assets or operations. A *threat* can originate from within the system (e.g., by an implementation error, configuration error, or from an external attacker). For example, the system can be configured to expose sensitive information without access control on a website [67]. This misconfiguration violates the security objective *confidentiality*. In contrast, an external attacker can perform a distributed denial-of-service attack to disrupt operation of a web server [183]. This attack then results in a violation of the security objective *availability*. Threat modeling [304] is an example of the application of design-time approaches to identify threats and to plan security measures to mitigate them.

Common threat categories are provided in threat modeling frameworks like STRIDE [304]. An example threat of STRIDE is *Information Disclosure*, which describes the exposure of sensitive information. This threat arises from the unallowed information flows in our running example where sensitive data (*userData* in the method *buy*) flows to insecure locations (*userData* in the method *store*). We use the threats of the STRIDE framework to populate our instance of the security model because they address the security objectives discussed before. The resulting instance is illustrated in Figure 6.3. However, when adding further security objectives, other threat categories can also be provided. For example,

if an instance of the security model is extended by the security objective *privacy*, the threats described by LINDDUN [278] can also be included. Some of these threats have an overlap with STRIDE such as *Information Disclosure* and *Linkability*. Thus, we propose to merge these threats to avoid semantically equivalent relations between security DSLs and security checks.

Weakness Threats are enabled by *Weaknesses* in the system. The NIST [279] defines a weakness as a flaw or characteristic that can lead to undesirable behavior. For example, a weak cryptographic algorithm could allow an adversary to read data without permission, such as the *userData* of the *buy* method in our running example (partial model (d) of Figure 3.2). As discussed in Subsection 5.2.2, the Common Weakness Enumeration (CWE) [327] provides a list of weaknesses which are known to arise in software systems. A single weakness can be further detailed by one or more weaknesses, which are represented in our security model as a `parent of` relationship. Capturing this relationship allows relating security checks and security DSLs through weaknesses of different levels of abstraction. For example, the weakness *CWE-328: Use of Weak Hash* is a more detailed instance of the weakness *CWE-326: Inadequate Encryption Strength*. We populated the instance of the security model used by *SecLan* with the entire CWE in version 4.12 [327]. Note that we use the term weakness in this context to refer to security issues that can be detected by security checks (see Subsection 2.3.1).

6.2.2. System Model

Design-time models and source code both describe systems. The elements of these models capture the system with abstractions, such as components in architectural models or classes in the implementation. Especially relevant for this dissertation are design-time models capturing the software architecture. To capture various system abstractions (e.g., business process models, architectural models and implementation), the *System Model* captures conceptual *System Element Types* (abbreviated with *Element Type*) relevant to security DSLs and source code analyzers. We also have identified relations between the *Element Types* necessary for relating security DSLs and security checks. Figure 6.4 illustrates the specific *Element Types* and relations between the *Element Types* we identified in our investigation of security DSLs and security checks.

The *Element Types* can be mapped to different system elements in DSLs and the source code. They do not represent concrete elements in a specific security DSL or in the source code but the concepts of these elements. For example, *methods* and *services* in our running example both represent an *activity* to be performed by the system. The *Semantic Relations* between the *Element Types* capture how these elements are related in a system. Additionally, we define the *If Compromised, also Compromised* relationship between elements to investigate how security issues on one *Element Type* could affect other *Element Types*.

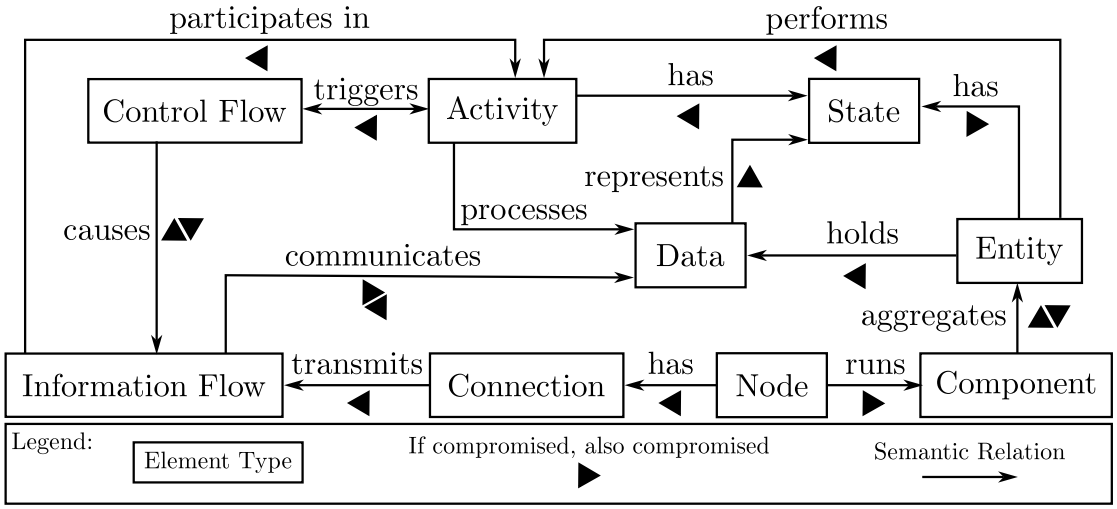


Figure 6.4.: System element types relevant in the context of security analyses and security DSLs

6.2.2.1. Element Types and Semantic Relationships

Here, we describe the semantics of the individual *Element Types* shown in Figure 6.4 and their semantic relationships. The coverage of specific *Element Types* varies between security DSLs and security checks because they address different concerns. Thus, in this section, we also present how often the specific *Element Types* are covered by the 66 security DSLs and 408 security checks we investigated.

Data A central element type in software security is *Data*, which conveys information through a collection of values. 63% of the security DSLs address *Data* (e.g., by labeling parameters with Security Characteristics in our running example in Figure 3.1). *Data* is directly analyzed by 32% of the checks.

Entity *Entities* can express a wide variety of concepts on different abstraction levels. We consider *Entities* to cover physical actors like users or attackers in a system, external systems such as databases, or low level elements such as software objects. An *Entity* can hold *Data*. For example, an object in the implementation is an *Entity* that holds *Data* via its attributes. Only 12% of the security checks analyze entities, but 85% of the security DSLs specify security-related information on them.

Activity System behavior is realized by *Activities*. *Activities* are performed by an *Entity* to process *Data* or to communicate with other *Activities*. The services of the architectural model and methods of the implementation in our running example are examples of *Activities*. Most security checks are performed on this element type (85%) and 67% of the security DSLs refer to it.

State *Entities* and *Activities* can have a *State* which is generally represented by *Data* (i.e., by the semantics of its values). Examples of *State* are the values of attributes of an object or the roles assigned to an adversary in a system. *States* are targeted by 39% of the security DSLs and 6% of the checks.

Control Flow The execution of *Activities* is orchestrated by *Control Flow*. 8% of the security DSLs refer to *Control Flow*. For example, UML activity diagrams [235] explicitly model conditional branches and loops as part of the *Control Flow*, while data flow diagrams contain it only implicitly. In the implementation, operation calls represent control flow and 14% of the security checks analyze them.

Information Flow An *Information Flow* describes the exchange of information between *Activities* or *Entities*. Commonly this exchange is realized by transferring *Data*. As described in Section 5.1, literature [20] describes information flow to comprise data flow and control flow. This distinction is captured in our system model by the *Information Flow* (1) communicating Data and (2) being caused by *Control Flow*. Since the security-relevant consideration is that transmitted data conveys information, we subsume both by the term information flow. 23% of security DSLs consider information flow for system description or security analysis. 44% of security checks address the leakage of sensitive information via techniques (e.g., in taint analysis [19, 135]).

Component Multiple software *Entities* can be combined to a *Component* that encapsulates specific functionality. Components have different granularity, ranging from an entire program to semantically related classes. In our running example, we present a special relationship between *Components* and *Entities* where the component is represented by a single class. This relationship resembles the consistency relationship proposed by Kramer et al. [173] which interpret the respective class as a facade for the functionality provided by a component. 35% of the security DSLs refer to components, while only 1% of checks investigate the implementation on the level of components.

Node A *Component* can be deployed on a *Node*, which is a physical device executing software. We acknowledge that *Node* is a broad term and can also refer to other domains in computer science like graph structures. There is also different terminology applied in literature for the semantic concept we address with *Node*. The PCM [275], for example, calls this concept *Resource Container*. Still, we remain with the term *Node* in line with deployment diagrams in UML [235] which is widely used in practice. While 21% of security DSLs address *Nodes*, no security check does so.

Connection Communication between *Activities* or *Entities* is established either through physical *Connections* between *Nodes* or internally within a component by inter-process communication. The physical *Connections* can be represented, for example, by *(W-)LAN*,

the Internet, local sockets or communication busses. Security DSLs refer more often to connections (25%) than security analyzers (3%).

6.2.2.2. The *If Compromised, also Compromised* Relationship

Attacks are typically realized by propagating through the system [348, 364]. Once an attacker compromises an element of the system, they may gain control over other parts of the system related to the compromised element. If an attacker in our running example compromises the communication link *Link1* (e.g., by performing Man-In-The-Middle attacks [53]) the attacker can read or manipulate the *Information Flow* over this link. Consequently, also the *Data* communicated over this link can be read or manipulated. Thus, the attacker can attempt to expand its influence from there by exploiting additional vulnerabilities. Accordingly, such propagation has to be considered when assessing the impact of a vulnerability. To support this assessment, the *SecLan* conceptual model captures the potential of propagation without the need to exploit a different vulnerability with the *If Compromised, also Compromised* relationship. Figure 6.4 illustrates this relationship by the triangular arrow on the semantic relations.

This relationship is not limited to one system abstraction, but spans design-time security and implementation-level security. Our pattern-search-based coupling approach described in Section 5.2 provides an example of this propagation across different system abstractions. An analyzer may detect a weak encryption algorithm used in an *Activity* in the implementation. The architectural analysis is not aware of the weak encryption as it does not capture *Activities*. However, the *Activity* compromises the *Data* which is intended to be encrypted and then transmitted in the *Information Flows* to other *Activities*.

Note that the *If Compromised, also Compromised* relationship in *SecLan* conceptual model does not differ from the *semantic relation* established in our pattern-search-based coupling approach in Section 5.2. Our pattern-search-based coupling approach (see Section 5.2) establishes a *semantic relation* between the *Activity Cipher.getInstance* which is affected by weak encryption, and the *Connection Link1* which is annotated with «*encrypted*» (see Figure 5.8). At a first look, this relationship does not exist in the *SecLan System Model* because there is no path over the *If Compromised, also Compromised* relationship between *Activity* and *Connection*. We establish this relationship in our pattern-search-based coupling approach due to the semantics of the «*encrypted*», expressing that *Data* transmitted over this link must be encrypted. Thus, while the annotation is performed on the *Connection*, it implies the *Data* in the *Information Flow* transmitted over this *Connection* has to be encrypted. This relationship is captured by the *If Compromised, also Compromised* relationships illustrated in Figure 6.4 between *Activity*, *Data* and *Information Flow*.

We systematically derived the *If Compromised, also Compromised* relationships from the observations of our study and related literature as described in Subsection 6.1.2. In the following, we give an overview of the relationships identified and the rationale behind them.

Node → Connection. In network security, attacks like *Denial of Service* or *Wormhole* [137] threaten the network itself. These attacks compromise a network *Node* to change network topography, flooding the network with messages, or modifying routing paths through the network [248]. Consequently, a network as a whole (i.e., its *Connections*) can be compromised if one of its *Nodes* is compromised. This is why we consider a *If Compromised, also Compromised* relationship between *Node* and *Connection*.

Connection → Information Flow. The Access Analysis [172], UMLSec [154], or the analysis of Almorsy et al. [12] evaluate whether communicated information is secure. They flag communicated information as insecure if the *Connection* over which the information flows is not correctly secured (e.g., by not employing encryption). Therefore, we describe a *If Compromised, also Compromised* relationship between *Connection* and *Information Flow*. A relevant type of attack illustrating this relation is man-in-the-middle attacks [53]: If an attacker can access the communication, the communicated information can be altered or read.

Node → Component The analyses of Walter et al. [348] and Hahner et al. [112] define that an attacker has access to all *Components* running on a *Node* which is compromised. This is why we define a *If Compromised, also Compromised* relationship from *Node* to *Component*.

Component ⇄ Entity Similarly to *Nodes*, if a *Component* is affected by a security issue, it is likely that the composed *Entities* are also affected. For example, in the case of a maliciously exchanged library, any *Entities* it contains (i.e., classes that provide functionality) must be considered compromised. Conversely, a compromised *Entity* can also affect the *Component* it is part of. For example, a virtual machine (*Entity*) that contains a VM escape vulnerability [13] can be exploited to compromise the host system (*Component*). Therefore, we define a bidirectional *If Compromised, also Compromised* relationship between *Component* and *Entity*.

Entity → Data, Entity → Activity Due to the differences in what can be expressed using *Entities*, we have to consider its relationships from two perspectives: an abstract perspective as considered in some security DSLs and a more narrow, low-level perspective as in the implementation. In social engineering, attackers compromise users (typically considered in design-time security DSLs by elements relating to the concept of an *Entity*) to obtain or manipulate *Data* like passwords [351]. Similarly, if a database (*Entity*) is accessible by an attacker (e.g., due to errors in permission management) the database content can also be obtained or altered [220]. Consequently, we define an *If Compromised, also Compromised* relationship between *Entity* and *Data*. Attackers can also use social engineering to convince users (*Entity*) to perform certain *Activities* (e.g., triggering a process) [351]. On the implementation-level, if a class has been compromised (e.g., via exploiting a deserialization vulnerability [251]), none of the *Activities* provided by this

Entity can be trusted. Consequently, we also define an *If Compromised, also Compromised* relationship between *Entity* and *Activity*.

Data → State As the *State* is defined by *Data*, if the *Data* can be modified by an attacker, the *State* can be equally modified. Consequently, we define an *If Compromised, also Compromised* relationship between *Data* and *State*.

State → Activity, State → Entity *CWE-642: External Control of Critical State Data* explicitly states that modification of a state can be used to perform "unauthorized actions (*Activity*) or access unexpected resources (e.g, *Data* held by *Entities*)" [332]. In a program, if executing an action depends on some state variable and an attacker can modify this variable (e.g., represented by a conditional branch), the attacker can decide whether the method can be executed. This is why we define a *If Compromised, also Compromised* relationship between *State* and *Activity* and *State* and *Entity*. Please note that a compromised *Activity* can also lead to a compromised state via compromised *Data* modified by the *Activity* that represents the *State*.

Data ⇄ Information Flow ⇄ Control Flow According to Balliu et al. [20], information flow analysis defines an information flow as a combination of *Control Flow* and *Data Flow*. If the *Data* in an information flow is compromised (e.g., because it can be manipulated by an attacker), then the information flow transporting this data is also compromised. Information flow analyzers like *JOANA* [98] flag other *Data* as tainted if they are affected by an already tainted information flow. Consequently, we describe a bidirectional *If Compromised, also Compromised* relationship between *Information Flow* and *Data*. Often, the control flow is influenced by data values. In our running example, the control flow is changed depending on whether the value of the shopping cart *cart.sum()* exceeds the limit of *privilegeSum*. If the information flow transporting this data can be manipulated, the control flow depending on this data is also compromised. Similarly, an information flow can be controlled if the *Control Flow* can be illegally modified. Therefore, we also define a bidirectional *If Compromised, also Compromised* relationship between *Information Flow* and *Control Flow*.

Activity → Control Flow, Activity → Information Flow. From the reasoning of the *If Compromised, also Compromised* between *Data*, *Information Flow* and *Control Flow*, it follows that if an *Activity* is compromised, then the *Control Flow* it triggers, or the *Information Flow* it participates in can be manipulated. If an attacker has access to the byte code of an *Activity*, they can modify the *Control Flow*, for example by inserting jump instructions to introduce security vulnerabilities [286]. Consequently, there is an *If Compromised, also Compromised* relationship between *Activity* and *Control Flow* and between *Activity* and *Information Flow*, respectively.

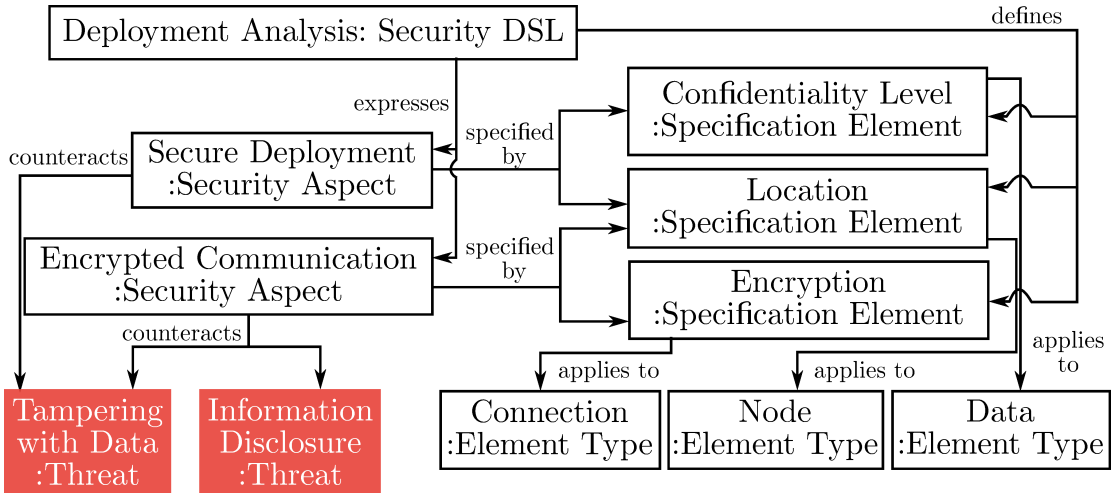


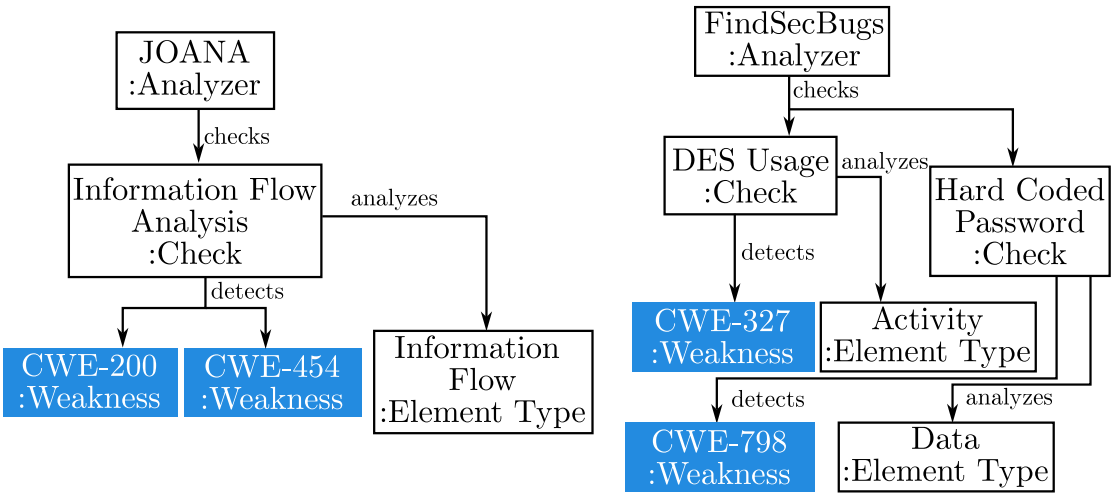
Figure 6.5.: Application of the *SecLan* conceptual model for the architectural analysis in our running example. The appendix *:X* indicates that the element is an instance of the concept *X* in the *Security DSL Description* illustrated in Figure 6.2. For legend regarding colored elements, see Figure 6.3

6.2.3. Security DSL Description

The *Security DSL Description* captures the concepts *Security DSL*, *Security Aspect* and *Specification Element* which are relevant to relate security DSLs to security checks. In general, an architectural analysis provides its own security DSLs to describe its input models or provides elements extending an existing DSL. The architectural analysis in our running example extends the PCM [275] by security-related concepts to describe expected security-related properties of the system architecture. Consequently, we expect the analysis architect to apply the *SecLan* conceptual model to describe the security DSL used in the architectural analysis. Because we use a black-box coupling, we only consider the input models and output models of the analyses in the coupling. Thus, describing the security DSL used in the architectural analysis is sufficient to establish the relation to potential analyzers to be used in the coupling. An instance of the *Security DSL Description* for the architectural analysis in our running example is illustrated in Figure 6.5.

Security DSL A *Security DSL* is used to design software systems to achieve one or more security objectives. For the purpose of performing design-time security analysis, security DSLs are specifically created to capture the system design and security-related information necessary to detect specific design-time security vulnerabilities.

Security Aspect *Security DSLs* define *Security Aspects* to counteract potential *Threats* from the security model. For example, the security DSLs for the architectural analysis in the running examples define the security aspect of *Secure Deployment* (illustrated in Figure 6.5). This security aspect aims to counteract *Information Disclosure* which can arise from processing confidential data on untrusted resources. *Security DSLs* can either define multiple *Security Aspects* to counteract different threats or focus on a single security aspect.



(a) *SecLan* conceptual model for the JOANA Analyzer

(b) *SecLan* conceptual model for the *Find Security Bugs* Analyzer (abbreviated with *FindSecBugs*)

Figure 6.6.: Application of the *SecLan* conceptual model to the source code analyzers *JOANA* and *Find Security Bugs* of our running example with example checks. For notation, see Figure 6.5. For legend regarding colored elements, see Figure 6.3.

Design-time security checks often investigate whether vulnerabilities related to these security aspects arise. Consequently, a vulnerability detected regarding a certain security aspect indicates that the corresponding threat is not sufficiently counteracted.

Related work like SecBPMN [288] map security aspects to the intended security objective (e.g., *integrity*). However, other related work [150, 159, 172, 338] structure security aspects more fine-grained (e.g., to provide aspects for secure information flow or secure physical access). In our analysis of 66 security DSLs, we identified counteracted threats as the abstraction applicable to all of them. These counteracted threats still establish the relation to the security objectives the security DSLs aim to protect.

Specification Element The *Specification Element* provides the means to specify the *Security Aspects* for a system. For this purpose, *Specification Elements* provide security-related information to *Element Types*. In our coupling approaches, we call these *Specification Elements Annotations* to highlight their role in the architecture and implementation. *Specification Elements* in our running example are the annotation of the *Confidentiality Levels* to *Activities* (i.e., services) and the annotation «*encrypted*» to *Connections*.

6.2.4. Security Analyzer Description

The *Security Analyzer Description* captures the concepts *Static Analyzer* and *Security Check*. These concepts are relevant to establish the relation of whether an analyzer can detect vulnerabilities that *can* result in non-compliance with the information described by a security DSL.

A *Security Check* is usually provided as part of an *Analyzer*. Analyzers like SonarQube [313] or CodeQL [100, 214] implement multiple security checks. In contrast, analyzers like JOANA [98] implement exactly one check. A *Security Check* inspects certain code elements, described by their *Element Type*, to detect vulnerabilities which are instances of predefined *Weaknesses*. Relevant weaknesses are either documented directly by analysis architects (e.g., in SonarQube check descriptions like [314]) or identified in academic papers (e.g., by Zhang et al. [366] for cryptographic *API Usage* analyzers). For analyzers where no weaknesses were documented, we searched the CWE [327] for relevant weaknesses. For the security analyzer VuRLE, we had to specify a wide variety of weaknesses since it is ML-based and has been trained on six entirely different types of vulnerabilities [196].

In contrast to our coupling approaches, we do not consider information in the *input models* of analyzers for *SecLan*. This information is only relevant to establish the capability that the vulnerabilities found by the analyzer can be used for the parameterization of the *Annotated Architecture* (see Chapter 5). However, the *Security Check* provides an *abstract* description of the vulnerabilities detected by the analyzer which are then represented in the *Source Code Analysis Result*. Consequently, as for the *Security DSL Description*, we expect the analysis architect of the source code analysis to apply the *SecLan* conceptual model to describe the respective analyzers. Because we only use the information in the *Source Code Analysis Result* for the coupling, these abstract descriptions provide a first indication of whether the vulnerabilities detected by the analyzer can lead to non-compliance with the security properties specified in the security DSL.

The application of the *SecLan* conceptual model to the analyzers *JOANA* [98] and *Find Security Bugs* [18] in our running example is illustrated in Figure 6.6. *JOANA* provides one security check *Information Flow Analysis* that analyzes *Information Flows* in Java programs. This check is described to detect the two weaknesses *CWE-200: Exposure of Sensitive Information to an Unauthorized Actor* and *CWE-454: External Initialization of Trusted Variables or Data Stores*. In contrast, *Find Security Bugs* provides multiple security checks. Two of these checks are illustrated in Figure 6.6. The first check inspects the usage of cryptographic APIs in Java programs (*Activity*) to detect weak encryption algorithms (*CWE-327: Use of a Broken or Risky Cryptographic Algorithm*). The second check inspects string literals (*Data*) to identify hard-coded passwords (*CWE-259: Use of Hard-coded Password*).

6.3. Relating Security DSLs to Source Code Analyses

Based on the concepts identified in the *SecLan* conceptual model, we aim to identify relationships between security DSLs and security checks. These relations describe that the security check can be applied to detect non-compliance between the implementation and information described in security DSLs. These relationships are established between the *Security Aspects* of the security DSLs and the *Security Checks* of the source code analyses.

We found that security checks and security aspects can be related using both the security domain and the system domain. These domains are captured in the *Security Model* and the *System Model* respectively. The relationship is established through the *Security Model* by the *Weaknesses* that are detected by security checks, and the *Threats* that are counteracted by *Security Aspects*. The relationship is established because the *Weakness* detected by the analyzer enables the *Threat* which is intended to be counteracted by the *Security Aspect*. The relationship is established through the *System Model* over the *Element Types* that are analyzed by *Security Checks* of the analyzers and that are enriched with security-related information by the *Specification Elements* of the security DSLs. Thus, *a check is related to a security aspect if and only if (1) at least one weakness detected by a security check of a security analyzer enables a threat counteracted by the security aspect, and (2) the system elements analyzed by the security checks and those targeted by specification elements are either identical or form a If Compromised, also Compromised relationship.*

For calculating the relationships between *Security Aspects* and *Security Checks*, we apply the concepts of graph-based model querying by using regular path expressions (see Subsection 2.1.4). Metamodels and models can be represented as graphs consisting of typed nodes and typed edges as described by Ebert and Bildhauer [74]. Ebert and Bildhauer calculate paths through these nodes using regular path expressions. These expressions describe sequences of edges between two nodes that have to exist so that a path is formed. The sequence is described based on the node types and the edges types connecting these nodes. With this expression, graphs representing models can be queried for paths between nodes of specific types connected by edges of specific types. We use these concepts to calculate the relationships between *Security Aspects* and *Security Checks* based on the relationships over the *System Model* and the *Security Model* described before.

For calculating relationships using path expressions, we represent the instances of the *SecLan* conceptual model and the provided *Element Types* as a graph. In this graph, every node represents either an instance of a concept in the *SecLan* conceptual model for a security DSL or analyzer or an *Element Type* we describe in Subsection 6.2.2. The nodes are typed with the type of the concept in the *SecLan* conceptual model or with the *Element Types*, respectively. Every edge in this graph represents a relation between the instances of two concepts or between two *Element Types*. The edges are typed with the type of the relation in the *SecLan* conceptual model or the type of the relation between the *Element Types*, respectively. Thus, each graph captures instances of *Security DSL Description*, instances of *Security Analyzer Description*, our instance of the *Security Model* (Figure 6.3) and the *Element Types* (Figure 6.4). An example graph for *JOANA* [98] in our running example is illustrated in Figure 6.6a. *JOANA* performs an information flow analysis to detect instances of the weakness *CWE-200*. In *SecLan*, we represent this information as two nodes: the first node is named *Information Flow Analysis* with the type *Security Check*, and the second node is named *CWE-200* with the type *Weakness*. Between these nodes, an edge of type *detects* exists.

We define a *Security Aspect a* and *Security Check b* to be *related* if there exist at least two paths from *a* to *b*: one path over the sub-graph representing the instance of the *Security Model* and another path over the sub-graph representing the *System Model* provided in

Subsection 6.2.2. However, there may be multiple paths over the *Security Model* and the *System Model*. Consequently, we call the combination of one path over the *Security Model* and one path over the *System Model* a *relationship* between the *Security Aspect a* and the *Security Check b*. We call the set of all relationships between a *Security Aspect a* and a *Security Check b* the *relation* between them.

We formalize these relationships by *related* in Equation 6.1 using regular path expressions (applying the notation discussed in Subsection 2.1.4). Note that the path expression is stated over the *types* of nodes and edges while the paths are calculated over the instances of the *SecLan* conceptual model (i.e., nodes and edges in the graph). Thus, applying *related* as a query over a graph containing an instance of the *SecLan* conceptual model with a *Security Aspect a* and a *Security Check b* yields all *relationships* between them.

$$\begin{aligned} \mathit{related}\colon a, b \mid \exists \mathit{path}(a \xrightarrow{\text{counteracts}} \xleftarrow{\text{enables}} \xrightarrow{\text{(parent of)}^*} \xleftarrow{\text{detects}} b) \wedge \\ \exists \mathit{path}(a \xrightarrow{\text{specified by}} \xrightarrow{\text{applies to}} \xleftarrow{\text{(if compromised, also compromised)}^*} \xleftarrow{\text{analyzes}} b) \\ \text{with} \quad a\colon \mathit{SecurityAspect} \quad b\colon \mathit{SecurityCheck} \end{aligned} \tag{6.1}$$

The first part of the expression describes the required path over the *Security Model*, while the second part describes the required path over the *System Model*. For the *Security Model*, we require the existence of a path that starts at the *Security Aspect a* over a *Threat* ($\xrightarrow{\text{counteracts}}$). This *Threat* must be enabled by a *Weakness* ($\xleftarrow{\text{enables}}$). This weakness must be detected by the *Security Check b* ($\xleftarrow{\text{detects}} b$). A path also exists if the weakness enabling the threat is a child of the weakness detected by the security check ($\xrightarrow{\text{(parent of)}^*}$). For the *System Model*, we require the existence of a path that starts at the *Security Aspect a* over a *Specification Element* ($\xrightarrow{\text{specified by}}$). This *Specification Element* must apply to an *Element Type* ($\xrightarrow{\text{applies to}}$). This *Element Type* must either be directly analyzed by the *Security Check b* ($\xleftarrow{\text{analyzes}} b$), or be reachable through the *If Compromised, also Compromised* relationship from an *Element Type* analyzed by the *Security Check* ($\xleftarrow{\text{(if compromised, also compromised)}^*} \xleftarrow{\text{analyzes}} b$).

Note how the length of the paths found over the *Security Model* and the *System Model* may vary. We express the length of a path as the number of edges traversed in the path obtained by applying the regular path expression. The minimal path length over the *Security Model* is three edges (illustrated in Equation 6.1): (1) from the *Security Aspect* to the *Threat*, (2) from the *Threat* to the *Weakness*, and (3) from the *Weakness* to the *Security Check*. Longer paths may exist if the detected *Weakness* is a child of the weakness enabling the threat. The minimal path length over the *System Model* is also three edges: (1) from the *Security Aspect* to the *Specification Element*, (2) from the *Specification Element* to the *Element Type*, and (3) from the *Element Type* to the *Security Check*. The length is exactly three edges if the *Element Type* analyzed by the *Security Check* is identical to the *Element Type* to which the *Specification Element* applies. Longer paths may exist if the *Element Type* to which the *Specification Element* applies is related to the *Element Type* analyzed by the *Security Check* via the *If Compromised, also Compromised* relationship. Consequently, the minimal length of a *relationship* between a *Security Aspect* and a *Security Check* is at least six edges.

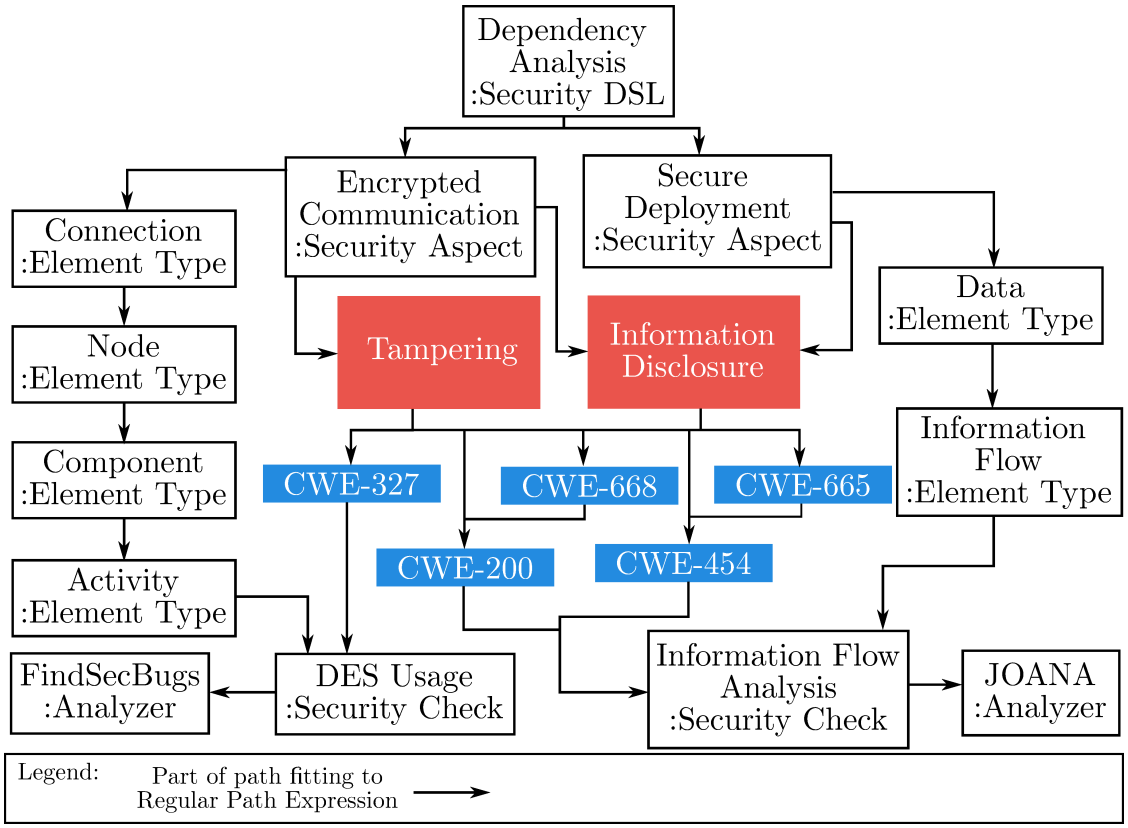


Figure 6.7.: Relations calculated by *SecLan* with the regular path expression in Equation 6.1 for the security DSLs of the architectural analysis and the source code analyzers of our running example (illustrated in Figure 6.5 and Figure 6.6 respectively). For legend regarding colored elements, see Figure 6.3.

Figure 6.7 shows exemplary relationships obtained when we apply *related* to the security DSLs instances (illustrated in Figure 6.5) and analyzer instances (illustrated in Figure 6.6) of *SecLan* for our running example. Note that the edges in Figure 6.7 show the paths calculated by the regular path expression and not the relations between the elements in the *SecLan* conceptual model. *JOANA* is related with the architectural analysis through the security aspect *Secure Deployment* and the security check *Information Flow Analysis*. The relationship over the *System Model* is established through the *Element Types Data* and *Information Flow*. The relationship over the *Security Model* is established because *JOANA* detects the weaknesses "*CWE-200: Exposure of Sensitive Information*" and "*CWE-454: External Initialization of Trusted Variables*". Further relationships exist over the parent-child relationships of the weaknesses. For example, "*CWE-668: Exposure of Resource*" is a parent of *CWE-200*, which could result in additional security objectives being threatened. These weaknesses enable the threats *Information Disclosure* and *Tampering with Data* which are counteracted by the security aspect *Secure Deployment*. *Find Security Bugs* is related to the architectural analysis through the security aspect *Encrypted Communication* and the security check *DES Usage*. This relationship is established over the *System Model* via the *If Compromised, also Compromised* relations between the *Element Types Connection*, *Node*, *Component* and *Activity*. The relationship is established over the *Security Model* because *DES Usage* detects the weakness "*CWE-327: Use of a Broken or Risky Cryptographic*

Algorithm" which enables both threats *Information Disclosure* and *Tampering with Data*. These threats are counteracted by the security aspect *Encrypted Communication*.

In Section 6.2, we describe that the *semantic relation* established in our pattern-search-based coupling approach in Section 5.2 does not contradict the relations in the *System Model*. As shown in Figure 6.7, *SecLan* detects a relationship between the security DSL of the architectural analysis and *Find Security Bugs* for the security check *DES Usage*. However, the relationships calculated by *SecLan* over the *System Model* (*Connection* →*Node* →*Component* →*Activity*) differ from the *System Elements* used in our semantic relation in our pattern-search-based coupling approach (*Activity* and *Data*). This is because *SecLan* does not consider *the semantics of the specification elements* when establishing relationships over the *System Model*. We discuss this limitation in Section 6.4.

The application of *SecLan* to our running example also shows the influence of the comprehensiveness of the *SecLan* conceptual model instance on the calculated relationships and the reasoning performed with them. Thus, the precision of *SecLan* is influenced by the analysis architect describing the source code analysis with the *SecLan* conceptual model. *SecLan is capable of* establishing relationships between the security DSL and *JOANA*. However, the reasoning about which capabilities of *JOANA* can be used to detect non-compliance is limited because the weakness *CWE-200* is broad. This is especially visible from the 13 CWEs that are children of *CWE-200*. Thus, while a relationship can be established, it is unclear whether *JOANA* is actually capable of detecting *Device Unlock Credential Sharing (CWE-1273)* or *Exposure of Sensitive Information Due to Incoming Policies (CWE-213)* which are children of *CWE-200*. We discuss this limitation in Section 6.4.

At first, establishing the relationships between the *Security Checks* and *Security Aspects* seems not to be equal to our coupling approaches. The coupling approaches require the coupling expert to establish the relation between the vulnerabilities found by the source code analysis and the *security characteristics of the coupling scenario* specified in the *Annotated Architecture* (see Chapter 5). In contrast, *SecLan* establishes relationships through the *Specification Elements* of the security DSLs. However, the *Specification Elements* describe the security-related information for the system element (i.e., they represent *Annotations* of *Security Characteristics* in the *Annotated Architecture*). Consequently, the coupling expert can investigate relationships calculated by *SecLan* to determine whether the *Specification Element* annotates the security characteristic in the coupling scenario. If a relationship between a security check and security aspect exists, and the specification element annotates the security characteristic in the coupling scenario, the coupling expert can investigate whether one of the coupling approaches described in Chapter 5 can be applied. For this, the coupling expert must evaluate whether the input and output models of the security DSLs and the analyzer for the security check conform to the reference metamodels (see Subsubsection 4.4.2.3), and whether either the ICs can be established (see Section 5.1) or the coupling model can be created (see Section 5.2).

6.4. Limitations

In this section, we discuss the limitations of *SecLan*. Future work to address these limitations is discussed in Section 11.3. We previously identified eleven limitations **L1** - **L11** of the coupling design and the coupling approaches in Section 4.5 and Section 5.3 respectively. We present four additional limitations **L12** - **L15** that arise in *SecLan*.

Limitation 12: Limited Data Sources We synthesized the *SecLan* conceptual model from literature on security DSLs and static analyzers. However, we cannot guarantee that our model covers all possible security DSLs or static analyzers. As a consequence, we cannot exclude that iterating over additional security DSLs and static analyzers may reveal other concepts. In turn, these concepts may require extending our *SecLan* conceptual model and could lead to other relationships calculated by *SecLan*. We addressed this limitation by selecting security DSLs and static analyzers from surveys, and by extending underrepresented categories (see Section 6.1). However, we acknowledge that *SecLan* can still be affected by this limitation.

Limitation 13: Referencing Abstract Weaknesses for Security Checks Leads to Imprecise Results The *SecLan* conceptual model must be instantiated for a specific security DSL (used in a design-time analysis) and static analyzer by the respective analysis architect. *SecLan* does not provide detailed guidance for instantiating the *SecLan* conceptual model. As we describe in Section 6.3, how *SecLan* depends on the *SecLan* conceptual model instances to calculate relationships between architectural analyses and source code analyses and to support the reasoning about this relationship. Incorrect or imprecise instantiations of the *SecLan* conceptual model lead to incorrect results. In particular, when the instantiation is imprecise, the results of *SecLan* may be complete but also imprecise and, consequently, not helpful for the coupling expert. We illustrated such a case by the instance of *JOANA* referencing *CWE-200: Exposure of Sensitive Information* instead of its children. Consequently, it is unclear for the coupling expert which vulnerabilities (and thus, which non-compliance) can be actually detected with *JOANA*. This challenge is highlighted in our evaluation of *SecLan* (see Chapter 9) where an expert stated that too many relationships were reported, presumably because of analyzers referencing too broad weaknesses.

Limitation 14: *SecLan* not considering Security Characteristics for Analysis Coupling The relationships calculated by *SecLan* provide first indications about potential source code analyses that can be coupled with a given architectural analysis. However, these relationships are calculated by considering the *Specification Elements* in the security DSLs rather than by considering the security characteristic addressed in a coupling scenario. Although the coupling expert can address this limitation by investigating every relationship, it may lead to missing potential security checks for the coupling. This challenge can increase the effort performed by the coupling expert if *SecLan* reports many relationships between the security DSL of an architectural security check and source code security checks.

Limitation 15: *SecLan* does not consider Semantics of Specification Elements We describe in Section 6.3 how the relationships calculated by *SecLan* between the *Specification Element Encrypted Communication* and the *Security Check DES Usage* differ from the *semantic relation* established in our pattern-search-based coupling approach in Section 5.2. This difference arises because *SecLan* does not consider the semantics of the *Specification Elements*. In our running example, the *Specification Element Encrypted Communication* is appended to the *Connection* (communication link). However, the semantics of this *Specification Element* imply that the *Data* transmitted over this *Connection* must be encrypted. Our *Element Types* provided in Subsection 6.2.2 also contain the relationship between *Activity* and *Data*, but this is not considered by *SecLan* when calculating relationships. In our example, *SecLan* detected a relationship over another path in the *System Model*. However, it is also possible that a relationship is missed due to this limitation if the annotated *Specification Element* affects only *Element Types* that are not reachable from the *Element Types* analyzed by the *Security Check*.

6.5. Summary And Outlook

In this chapter, we presented *SecLan* which provides a systematic process to calculate relationships between security DSLs and static source code analyzers as our third contribution **C3** to answer ❓ **Research Question 4**. The relationships indicate which security check of a source code analyzer is potentially capable of detecting non-compliance with security properties specified in a security DSL. *SecLan* addresses ❌ **Problem 3** by supporting coupling experts in selecting security checks that are capable of detecting non-compliance with information specified in a security DSL. Relating the source code security checks and their analyzers with security DSLs is feasible for supporting the coupling expert because security DSLs are commonly used to define the input models of the architectural security analyzer.

In the third contribution **C3**, we first introduced our method to synthesize the *SecLan* conceptual model from 66 security DSLs and 33 static source code security analyzers comprising 408 security checks. We then present the *SecLan* conceptual model which captures concepts and relations between these concepts for the purpose of relating security DSLs and security checks provided by static source code security analyzers. Based on the *SecLan* conceptual model, we defined how to calculate relationships between a security DSL and security checks performed by a source code security analyzer. Finally, we discussed limitations of *SecLan*.

Method for Synthesis of the *SecLan* Conceptual Model We performed a systematic process to synthesize the *SecLan* conceptual model from 66 security DSLs and 33 static source code security analyzers comprising 408 security checks. This process involved selecting security DSLs and static source code security analyzers from surveys, and extending underrepresented categories. We initialized the *SecLan* conceptual model with concepts and relations identified from analyzed security DSLs and static source code security

analyzers, expert knowledge, and literature. We then iteratively analyzed the selected security DSLs and static source code security analyzers to identify concepts and relations between these concepts common to both security DSLs and static source code security analyzers. In this process, we classified the security DSLs and static source code security analyzers into the categories discovered. If a new category was discovered, we discussed new concepts and relations to be added to the *SecLan* conceptual model with collaborators and updated the *SecLan* conceptual model accordingly. If the *SecLan* conceptual model was updated, we re-iterated over the selected security DSLs and static source code security analyzers to identify whether the new concepts and relations are also present in other security DSLs and static source code security analyzers. Once no new concepts and relations were discovered, we finalized the *SecLan* conceptual model.

***SecLan* conceptual model** The *SecLan* conceptual model captures concepts and relations between these concepts for the purpose of relating security DSLs and security checks provided by static source code security analyzers. We separated the *SecLan* conceptual model into four sub-models: the *Security Model*, the *System Model*, the *Security DSL Description* and the *Security Analyzer Description*. The *Security Model* and the *System Model* capture general concepts and relations between these concepts to represent the security domain and system domain respectively. The *Security DSL Description* and the *Security Analyzer Description* capture concepts specific to security DSLs and static source code security analyzers respectively.

For each sub-model, we defined the concepts and relations between these concepts. We instantiated the *Security Model* with the STRIDE model [304]. For the *System Model*, we defined specific *Element Types* (representing types of system elements) and relations between them used in *SecLan*.

Relationships between security DSLs and Static Source Code Security Analyzers Based on the *SecLan* conceptual model, we define how we calculate relationships between a security DSL and security checks performed by a source code security analyzer. For this purpose, we represent all instances of the *SecLan* conceptual model describing the security DSLs and the source code security analyzers as a combined graph. In this graph, the nodes represent instances of the concepts from the *SecLan* conceptual model (e.g., *Security Aspects*, *Security Checks* or *Element Types*) and edges represent relations between them. The nodes are typed with the concepts, and the edges typed with the type of relation from the *SecLan* conceptual model. We define a regular path expression [74] over this graph which describes when a security check is related to a security DSL. Related in this context means that the vulnerabilities detected by the security check violate security properties specified in a security DSL. For this relationship, we define two paths in the combined graph: one over the *Security Model* and one over the *System Model*. The result of a query with this regular path expression is a set of pairs of security checks and security DSLs which are related according to the defined regular path expression.

Limitations *SecLan* is subject to several limitations. Most notably, relationships calculated by *SecLan* only provide preliminary guidance on whether an analyzer can be used in the coupling with an architectural analysis because *SecLan* does not consider security characteristics under investigation in a coupling scenario and the semantics of specification elements when establishing relationships.

Outlook This chapter (Chapter 6) presenting *SecLan* concludes our contributions in the area of coupling architectural security analyses with static source code security analyses in this thesis. We continue with *Part III - Validation* of this thesis where we present the evaluation of our contributions. In the next chapter (Chapter 7), we present the evaluation goals and data collection methods applied to evaluate our three contributions **C1**, **C2** and **C3**. Chapter 9 presents the evaluation of *SecLan*.

Part III.

Validation

7. Evaluation Objectives and Data Collection Methods

Literature: This chapter is based on the following (co-) authored publications: [IEEE TSE 2025], [IEEE ICSA-C 2025], and "*Can I Check What I Designed? Mapping Security Design DSLs to Code Analyzers*" [254] (major revision under review in the journal *ACM Transactions on Software Engineering and Methodology* at the date of submission of the dissertation)

This chapter presents the evaluation objectives [282] and the data collection methods [311] we apply in evaluating contributions **C1**, **C2**, and **C3**. According to Runeson and Höst [282], the evaluation objectives define *what* is evaluated. The *Data Collection Methods* [311] define how data is collected (e.g., by performing case studies, surveys, or expert interviews).

We evaluate our contributions **C1** and **C2** (the coupling design and coupling approaches) quantitatively. In contrast, our evaluation of **C3** (*SecLan*) contains both quantitative and qualitative parts. The quantitative parts of the evaluation are structured using the Goal-Question-Metric (GQM) approach of Basili et al. [23]. In this approach, first goals that have to be reached in the evaluation are defined. Then, questions are asked to determine whether the goals are achieved. Finally, metrics are defined which allow us to investigate whether the questions are answered positively or negatively. We state the goals, questions, and metrics in Section 7.1. We discuss the applied data collection methods and why their application is reasonable in Section 7.2. This discussion is especially relevant because some metrics require specific types of data (e.g., expert feedback) which are obtained by applying specific kinds of data collection methods.

In the context of evaluations using case studies, Runeson and Höst [282] combines the evaluation objectives with the detailed planning of how to collect data in a *Case Study Design*. We separate the *evaluation objectives* from the details on *how* we collect the data (hereafter called *study design*). This separation allows us to provide an overview of the validation structure before presenting details of the specific evaluations of our contributions. We do this because details about *how* the data is collected are necessary to provide a comprehensive evaluation. However, these details differ for various data collection methods. For example, when collecting data with a case study, these details include, among others, defining the cases and how the metrics are calculated based on the data collected in the cases [282]. In contrast, collecting data with surveys includes, for example, details such as the survey questions and how to recruit participants. The collected data must then be analyzed, and the obtained insights discussed [166, 282].

Part III - Validation of this thesis is structured into three chapters. This chapter (Chapter 7) presents the evaluation objectives and data collection methods to structure the validation of our contributions. The second chapter (Chapter 8) presents the study design and results in the evaluation of the coupling design and coupling approaches (**C1** and **C2**). The third chapter (Chapter 9) presents the study design and results in the evaluation of *SecLan* (**C3**). We combine the evaluation of the coupling design and coupling approaches (**C1** and **C2**) because they are closely related contributions — the coupling approaches instantiate the coupling design. Specifically, without a coupling approach, the coupling design is not instantiated and cannot be empirically evaluated. In contrast, *SecLan* (**C3**) is an independent contribution which can be evaluated without considering the other contributions. Thus, the evaluation objectives presented in this chapter provide an overview of *what* is evaluated in the remaining chapters of *Part III - Validation*. This structure helps the reader to understand the overall evaluation and allows for better navigation in *Part III - Validation* of this thesis.

7.1. Evaluation Goals, Questions and Metrics

In this section, we describe the *goals* of our evaluation (i.e., evaluation objectives), the *questions* asked to determine if we achieve these goals and the *metrics* to determine whether the questions can be answered positively or negatively. We state the goals in a form that a capability of the respective contribution can be evaluated. If the goal is not met, the contribution does not provide the respective capability. We start by describing the goals, questions and metrics to evaluate the coupling design and coupling approaches (**C1** and **C2**). Then, we describe the goals and questions to evaluate *SecLan* (**C3**).

In this section, the following notation for goals, questions, and metrics is applied. We sequentially enumerate the goals used to evaluate our contributions, i.e., **G1**, **G2**, **G3** and **G4**. The questions are denoted by (1) a first number that is equal to the number of the assigned goal, and (2) a sequentially increasing number within the goal. An example for this scheme is the first question **Q1.1** assigned to the goal **G1**. We use a similar scheme for assigning metrics to questions: (1) the first two numbers are the ones of the question the metric is assigned to, and (2) a sequentially increasing number within this question. An example for this scheme is the first metric **M1.1.1**, assigned to question **Q1.1**. However, some metrics are applied to multiple questions, if the questions address similar aspects of the evaluated contribution. In this case, the metric number starts with the number of the *first* question the metric is assigned to. This case can be observed, for example, with **M1.1.1** which is assigned to both questions **Q1.1** and **Q1.2**.

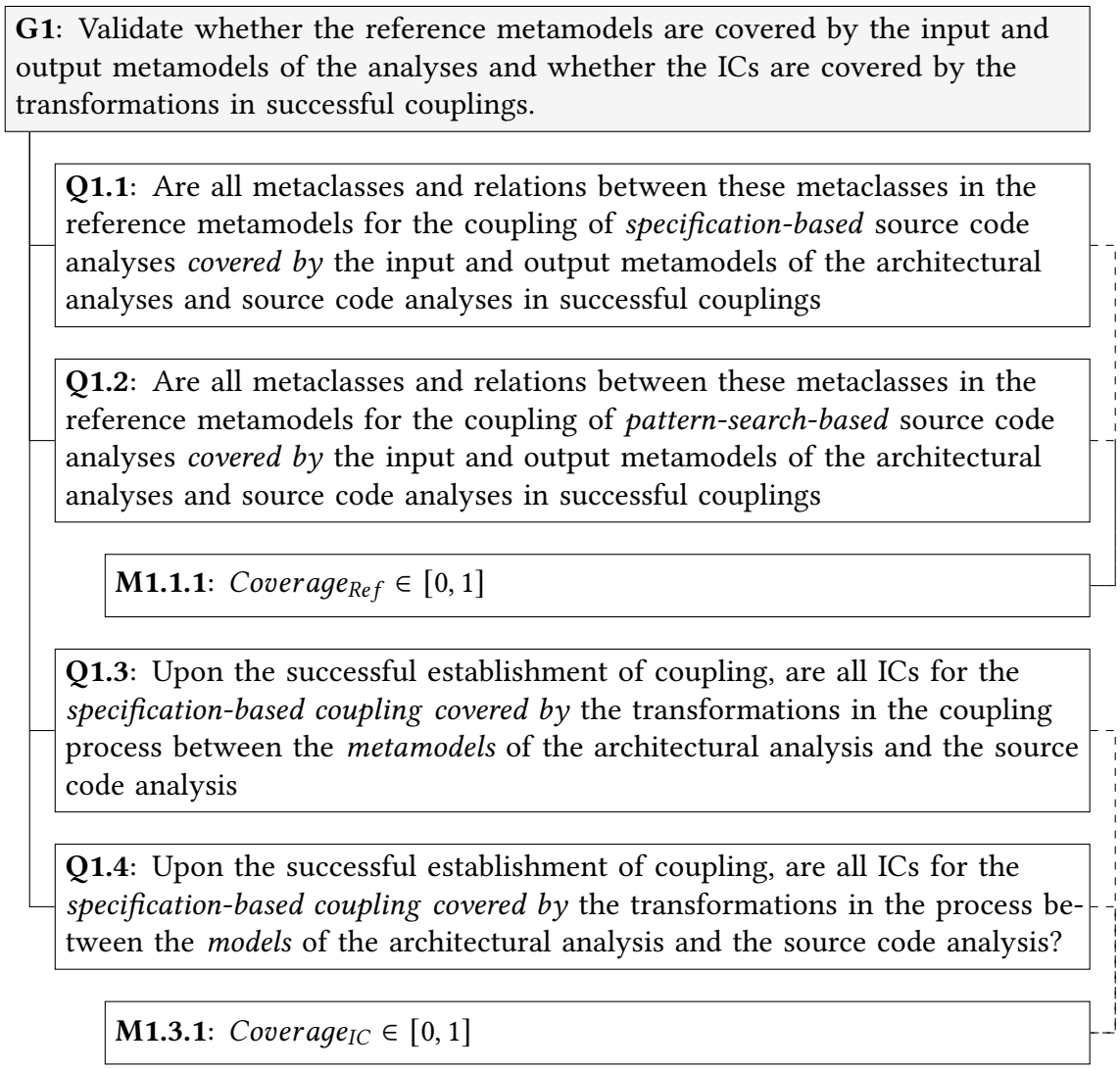


Figure 7.1.: Overview of the GQM plan for evaluating the coverage of reference metamodels and Integration Conditions (ICs).

7.1.1. Goals, Questions and Metrics to Evaluate Coupling Design and Coupling Approaches

We evaluate **C1** and **C2** with the first two evaluation goals **G1: Coverage of Reference Metamodels and ICs** and **G2: Accuracy of Coupled Analysis**.

7.1.1.1. Evaluation Goal 1: Coverage of Reference Metamodels and Integration Conditions

We start by evaluating the reference metamodels of the specification-based coupling approach and pattern-search-based coupling approach. Additionally, we evaluate the Integration Conditions (ICs) of the specification-based coupling approach. We do not provide ICs for the pattern-search-based coupling approach as explained in Subsection 5.2.3.

The reference metamodels are described as interfaces for the coupling (see Chapter 5). A successfully established coupling without conformance of the input and output metamodels of the coupled analyses to the reference metamodels would contradict the assumption about the reference metamodels defining interfaces. Similarly, the ICs in the specification-based coupling approach describe *necessary* but not *sufficient* conditions which define required relations between the metamodel and models of the analyses in the coupling (again, see Chapter 5). A successfully established coupling where the relations described by the ICs do not exist would imply that the respective ICs are not necessary. Consequently, we begin evaluating **C2** by setting the first goal **G1**:

> **G1: Coverage of Reference Metamodels and Integration Conditions** Validate whether the reference metamodels are covered by the input and output metamodels of the analyses and whether the ICs are covered by the transformations in successful couplings.

The GQM-Plan for goal **G1** is summarized in Figure 7.1. We start by evaluating the coverage of the reference metamodels because they build the foundation for the ICs. If the reference metamodels are not covered by the metamodels of the analyses in successful couplings, the ICs cannot be covered either.

For each metaclass in the reference metamodels, the input and output metamodels of the analyses must provide at least one *conforming* metaclass to be applicable in our coupling (see Subsubsection 4.4.2.3). If a metaclass in the reference metamodels would not be covered by input and output metamodels of the analysis in a successful coupling, the metaclass is not *necessary* for the coupling. Interfaces that comprise metaclasses not necessary for the coupling would potentially result in the coupling expert to wrongly dismiss analyses as they do not comprise the necessary elements in their metamodels.

We call a metaclass in the reference metamodels *covered* by the input and output metamodels of the analyses, if at least one conforming metaclass exists in these metamodels. Consequently, we call the reference metamodels *covered* by the input and output metamodels of the analyses, if all metaclasses in the reference metamodels are covered by these metamodels. With this definition, we ask the first and second questions about the coverage of the reference metamodels for the specification-based coupling (**Q1.1**) and for the pattern-search-based coupling (**Q1.2**):

> **Q1.1** Are all metaclasses and relations between these metaclasses in the reference metamodels for the coupling of *specification-based* source code analyses *covered by* the input and output metamodels of the architectural analyses and source code analyses in successful couplings?
>
> **Q1.2** Are all metaclasses and relations between these metaclasses in the reference metamodels for the coupling of *pattern-search-based* source code analyses *covered by* the input and output metamodels of the architectural analyses and source code analyses in successful couplings?

The reference metamodels build the foundation for our ICs in the specification-based coupling approach. These ICs enable the parameterization of the *Annotated Architecture* by vulnerabilities detected with a static source code security analysis. We describe these ICs as *necessary but not sufficient* conditions that establish relations between the metamodels (and their models) of the analyses. Similarly to the metaclass in the reference metamodels, a IC would be unnecessary for the coupling, if a coupling is established successfully, but the condition does not hold (i.e., the relations required do not exist). We formulate the ICs to hold on the metamodel and on the model level. Covering both levels is important due possible model-level semantic refinements (see Subsection 4.3.3). Therefore, we investigate the coverage of the ICs for the *specification-based coupling approach* on the metamodel level (**Q1.3**) and on the model level (**Q1.4**):

> **Q1.3** Upon the successful establishment of coupling, are all ICs for the *specification-based coupling covered by* the transformations in the coupling process between the *metamodels* of the architectural analysis and the source code analysis?
>
> **Q1.4** Upon the successful establishment of coupling, are all ICs for the *specification-based coupling covered by* the transformations in the process between the *models* of the architectural analysis and the source code analysis?

Related work [39, 157, 171, 221] evaluates coverage similarly to our four questions **Q1.1**, **Q1.2**, **Q1.3**, and **Q1.4**. Muñoz Hurtado et al. [221] inspect how many measured API endpoints cover secured operations defined in a security requirements specification. Bucaioni et al. [39] provide a metric to capture how many components in a specific architecture *conform* to a reference architecture. Kaplan et al. [157] evaluate the *soundness* of a taxonomy by inspecting whether the taxonomy is covered by given sentences. Konersmann et al. [171] investigate whether a reference model is *realized* by specific models by investigating for each model element in the reference metamodel whether a *semantically valid* mapping to the specific model can be defined. While these related works determine whether some artifacts are realized by other artifacts, they refer to this capability differently. The difference between these related evaluations and ours is this: related work calculates the metrics between specific artifacts, which are tailored to the respective evaluation and the artifacts under study. For example, Konersmann et al. [171] define a metric specifically tailored to evaluate whether a reference model is realized by specific models. In contrast, we aim to evaluate the coverage between different artifacts, i.e., (1) between the reference metamodels and the input and output metamodels of the analyses, and (2) between the ICs and the realized transformations in the coupling process.

Thus, for a more comprehensive evaluation, we generalize the questions **Q1.1**, **Q1.2**, **Q1.3**, **Q1.4**, and the metrics of related work to the metric *Coverage* in Equation 7.1.

$$Coverage(A, COND) = \frac{\sum_{cond \in COND} hold(A, cond)}{|COND|} \tag{7.1}$$
$$\text{with} \quad hold\colon A \times COND \rightarrow \{0, 1\}$$

Coverage calculates the degree to which conditions *cond* in the set of all conditions *COND* hold for given evaluation artifacts A. For operationalization, we define an evaluation

function *hold*, which determines for the given set of evaluation artifacts A whether a condition *cond* does or does not hold. If a condition holds, we denote the condition as *covered*. Consequently, to answer a specific evaluation question, *Coverage* must be specialized by defining the conditions (*COND*), the evaluation artifacts (A), and the evaluation function *hold*.

Thus, to evaluate the coverage of the reference metamodels (**Q1.1** and **Q1.2**) and the coverage of the ICs (**Q1.3**, **Q1.4**), we specialize Equation 7.1 accordingly. We denote the *Coverage* metric for the reference metamodels (**Q1.1** and **Q1.2**) with **M1.1.1** $Coverage_{Ref}$. Similarly, we denote the *Coverage* metric for the ICs (**Q1.3**, **Q1.4**) with **M1.3.1** $Coverage_{IC}$. The coverage of the reference metamodel is evaluated for both coupling approaches separately because the reference metamodels differ between the two approaches. To allow for better distinction when discussing the results for this metric, we denote the *Coverage* of the reference metamodels for the specification-based coupling approach and for the pattern-search-based coupling approach as $Coverage^{S}_{Ref}$ and $Coverage^{P}_{Ref}$ respectively. The conditions *COND* and the evaluation function *hold* for the *Coverage* of the reference metamodels for the pattern-search-based coupling approach and specification-based coupling approach are defined similarly. Consequently, we define the conditions *COND* and the evaluation function *hold* for both coupling approaches combined in Chapter 8. The conditions *COND* and *hold* must be applied to the evaluation artifacts A. These artifacts are specific to the respective coupling approaches. For example, because the source code analyses in the two coupling approaches differ, the input and output metamodels of these analyses also differ. Consequently, we define the evaluation artifacts A for both coupling approaches in their individual study designs in Chapter 8. We also define the conditions *COND*, the evaluation function *hold*, and the evaluation artifacts A for the *Coverage* of the ICs in Chapter 8.

7.1.1.2. Evaluation Goal 2: Accuracy of Coupled Analysis

The goal of our analysis coupling is to detect architectural vulnerabilities arising from implementations that are non-compliant with the security characteristic of the coupling scenario. However, if the coupled architectural analysis does not properly detect such vulnerabilities, the coupling does not provide any benefit compared to the effort arising to establish it. Additionally, if the coupled architectural analysis provides too many false alarms, software engineers may neglect the analysis results [151, 287] and even face an increase in worthless effort [103]. Consequently, we evaluate **C1** and **C2** by setting the second goal **G2**.

G2: Accuracy of Coupled Architectural Analysis Validate the accuracy of the coupled architectural analysis regarding the detection of architectural vulnerabilities arising from an implementation non-compliant with the security characteristic of the coupling scenario.

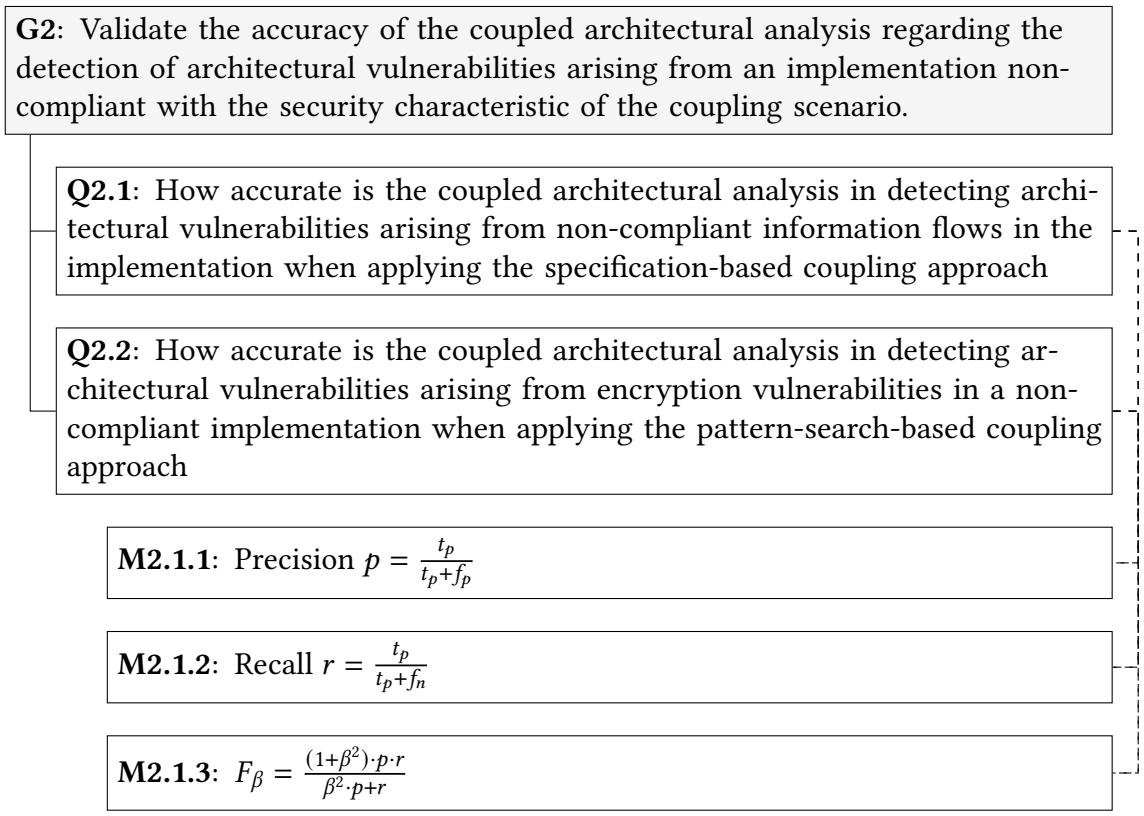


Figure 7.2.: Overview of the GQM plan for evaluating the accuracy of the coupled architectural analysis regarding the detection of architectural vulnerabilities arising from a non-compliant implementation.

The GQM-Plan for goal **G2** is summarized in Figure 7.2. The accuracy of the architectural analysis is supposed to be improved by the coupling, when compared to isolated architectural analysis. This is because the architectural analysis in the coupling is intended to detect vulnerabilities originating from the non-compliant implementation. However, the isolated architectural analysis is incapable of detecting such vulnerabilities. Therefore, it is not possible to compare the isolated architectural analysis to the coupled analysis. For this reason, we investigate how accurately the coupled architectural analysis detects vulnerabilities that arise from a non-compliant implementation regarding the values of security characteristics of a coupling scenario.

Based on this premise, we ask question **Q2.1** and **Q2.2** about the accuracy of the coupled architectural analysis for the specification-based coupling approach and the pattern-search-based coupling approach respectively:

Q2.1 How accurate is the coupled architectural analysis in detecting architectural vulnerabilities arising from non-compliant information flows in the implementation when applying the specification-based coupling approach?

Q2.2 How accurate is the coupled architectural analysis in detecting architectural vulnerabilities arising from encryption vulnerabilities in a non-compliant implementation when applying the pattern-search-based coupling approach?

In related work (e.g., [12, 19, 250, 300, 348]), the metrics commonly used to evaluate the accuracy of analyses are precision p (**M2.1.1**) and recall r (**M2.1.2**), both illustrated in Equation 7.2. Precision p and recall r are calculated by considering true positives t_p, false positives f_p, and false negatives f_n in the results of the analysis under study (i.e., the coupled architectural analysis in this dissertation).

$$\text{Precision } p = \frac{t_p}{t_p + f_p} \qquad \text{Recall } r = \frac{t_p}{t_p + f_n} \tag{7.2}$$

Precision p represents the number of correct reports of vulnerabilities by the coupled architectural analysis compared to all vulnerabilities reported. Investigating precision p is essential to understand the degree to which the coupling produces false alarms (false positives f_p). Recall r represents the number of correct vulnerabilities found compared to all vulnerabilities existing in the system. Thus, investigating recall r is essential to understand the degree to which the coupling misses relevant vulnerabilities (false negatives f_n).

In the domain of security, false alarms are often preferred to missed detections because missed detections may enable successful attacks by leaving vulnerabilities undetected [14]. However, a high number of false alarms may also lead software engineers to neglect an analysis [151, 287]. This leads to the conclusion that both precision p and recall r are essential metrics to evaluate the accuracy of the coupled analyses. To allow for precise comparison of the accuracy of further coupling approaches, we also use the F_β-Score (**M2.1.3**) used in related work [65, 111] as shown in Equation 7.3.

$$F_\beta = \frac{(1 + \beta^2) \cdot p \cdot r}{\beta^2 \cdot p + r} \tag{7.3}$$

The F_β-Score combines precision p and recall r into a single metric which enables comparability. The parameter β results in weighting the recall r higher than precision p (for $\beta > 1$) or vice versa (for $\beta < 1$). A common β value is $\beta = 1$ [65, 111] which results in the harmonic mean between precision p and recall r. Berry [28] discusses in the domain of requirements engineering and natural language processing that choosing β depends on the application context. Díaz and Bermejo [65] state that $\beta = 2$ is another commonly used value to weight recall higher than precision. This value aligns with the statement that missing a security vulnerability is often considered more severe than a false alarm [14]. Thus, we decide to calculate the F_β-Score for $\beta = 1$ and $\beta = 2$. However, we are aware that the most appropriate β should be evaluated empirically in future work.

Precision and recall can be reasonably evaluated only if the results expected from an analysis are already known *and* only if a comprehensive definition of t_p, f_p or f_n is available. Therefore, we define our ground truth (i.e., the analysis results expected) and the classification of the actual results into t_p, f_p or f_n likewise in our study design for evaluating **C1** and **C2** in Chapter 8.

7.1.2. Goals, Questions and Metrics to Evaluate *SecLan*

We evaluate *SecLan* (**C3**) with two evaluation goals **G3: Correctness and Completeness of *SecLan* conceptual model** and **G4: Qualitative Evaluation of Relations Calculated by SecLan**.

7.1.2.1. Evaluation Goal 3: Completeness and Correctness of *SecLan* Conceptual Model

To calculate the relations in *SecLan*, the *SecLan* conceptual model must be instantiated to represent security DSLs and analyzers. The *SecLan* conceptual model cannot be reasonably applied if it does not contain elements necessary to represent a broad range of security DSLs and analyzers (referred to as *Completeness* by Bertoa and Vallecillo [29]). Also, the *SecLan* conceptual model cannot be reasonably applied if the provided concepts and relations between the concepts are defined wrongly regarding the domain they are applied in (referred to as *Correctness* by Bertoa and Vallecillo [29]). Thus, we begin evaluating the *SecLan* conceptual model as part of contribution **C3** by setting the third goal **G3**.

> **G3: Completeness and Correctness of *SecLan* conceptual model** Validate whether the *SecLan* conceptual model can be applied to represent security DSLs and analyzers correctly and completely

The GQM-Plan for goal **G3** is summarized in Figure 7.3. We synthesized the concepts in the *SecLan* conceptual model from literature by applying our research method outlined in Section 6.1. However, *SecLan* can miss relations between *Security Aspects* and *Security Checks* if the synthesized concepts and relations between these concepts are not *complete*. Similarly, *SecLan* can determine false relations between *Security Aspects* and *Security Checks* if the synthesized concepts and relations between the concepts in the *SecLan* conceptual model are incorrect. We define *correctness* in this context so that the semantics of the concepts and the semantics of the relations between these concepts is consistent with the knowledge in the domains of security DSLs and analyzers. *Correctness* can be violated, for example, because we misinterpreted the data sources while extracting the concepts and relations.

We define *completeness* in this context so that all concepts and relations between these concepts exist in the *SecLan* conceptual model necessary to represent security DSLs and analyzers. *Completeness* can be violated, for example, if we excluded concepts or relations because they were only sparsely represented in the data sources used. Similar to the concepts in the *SecLan* conceptual model, we extracted the specific *Element Types* and the relations (*semantic relations* and *If Compromised, also Compromised* relationships) between the *Element Types* in the *System Model* from the data sources. Thus, also the specific *Element Types* have to be validated for correctness and completeness as they build the foundation for calculating the relations between *Security Aspect* and *Security Checks*. We include the *System Model* as a dedicated model due to the number of different, more fine-granular predefined concepts and relations between these concepts provided in it.

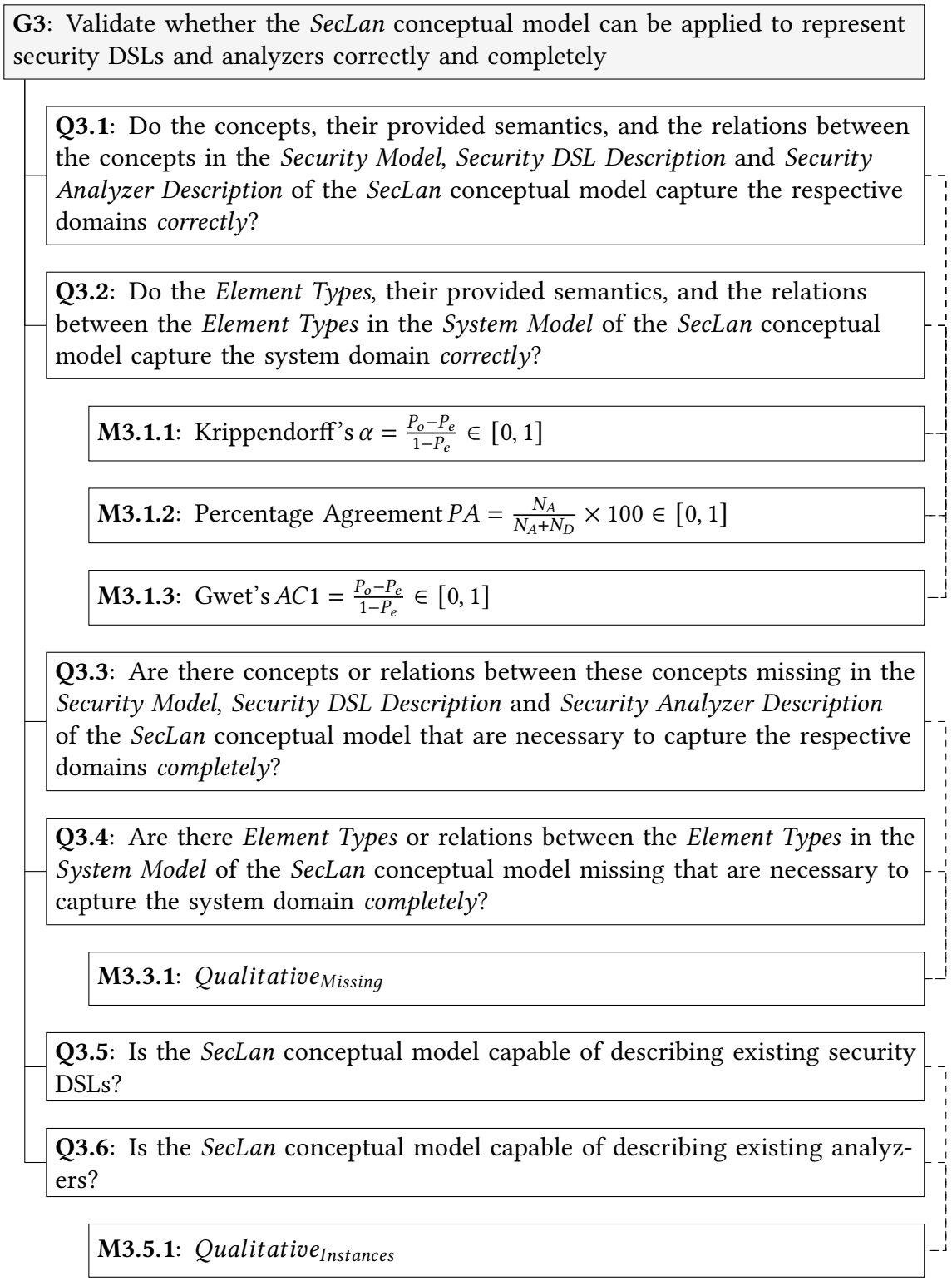


Figure 7.3.: Overview of the GQM plan for evaluating the correctness and completeness of the *SecLan* conceptual model.

We ask the questions **Q3.1** and **Q3.2** about the *correctness* of the concepts, *Element Types* and their respective relations in the *SecLan* conceptual model. We also ask the questions **Q3.3** and **Q3.4** about the *completeness* of the concepts, *Element Types* and their respective relations in the *SecLan* conceptual model.

> **Q3.1** Do the concepts, their provided semantics, and the relations between the concepts in the *Security Model*, *Security DSL Description* and *Security Analyzer Description* of the *SecLan* conceptual model capture the respective domains *correctly*?
>
> **Q3.2** Do the specific *Element Types*, their provided semantics, and the relations between the *Element Types* in the *System Model* of the *SecLan* conceptual model capture the system domain *correctly*?
>
> **Q3.3** Are there concepts or relations between these concepts missing in the *Security Model*, *Security DSL Description* and *Security Analyzer Description* of the *SecLan* conceptual model that are necessary to capture the respective domains *completely*?
>
> **Q3.4** Are there specific *Element Types* or relations between the *Element Types* in the *System Model* of the *SecLan* conceptual model missing that are necessary to capture the system domain *completely*?

To evaluate whether the extracted concepts and relations between these concepts in the *SecLan* conceptual model (**Q3.1**) and the specific *Element Types* in the *System Model* (**Q3.2**) correctly capture the represented domains, we perform a survey with experts. In this survey, we assess whether experts agree with the concepts and relations between these concepts in the *SecLan* conceptual model. We discuss the necessity of expert agreement for evaluating *correctness* in Section 7.2.

We aim to summarize the overall agreement between experts about the correctness of the *SecLan* conceptual model and the specific *Element Types* in the *System Model*. An established method to capture the agreement between experts is to calculate inter-rater reliability metrics [175, 193]. These metrics indicate the degree of agreement between experts. A prominent metric is *Krippendorff's Alpha* α [175]. Krippendorff's Alpha is capable of considering more than two experts, binary data (e.g., the agreement or disagreement to our concepts, *Element Types* and relations between them) and considers agreement occurring by chance. Other metrics such as Cohens Kappa are only applicable for two experts [175]. Thus, we calculate Krippendorff's α (**M3.1.1**) to assess the agreement between multiple experts about our concepts and relations between the concepts in the *SecLan* conceptual model (**Q3.1**) and about the *Element Types* and relations between them (**Q3.2**). The high-level formula for Krippendorff's α is shown in Equation 7.4.

$$\alpha = \frac{P_o - P_e}{1 - P_e} \tag{7.4}$$

In Equation 7.4, P_o describes the relative observed agreement among experts and P_e describes the probability of chance agreement. P_e is inferred from the observed data [175]. The value of α ranges from -1 to 1, where 1 indicates perfect agreement, 0 indicates no

agreement beyond chance, and negative values indicate less than chance agreement [175]. P_o and P_e are calculated by investigating a matrix where the rows represent the coders (experts) and the columns represent the units they rate. In our case, the units are the concepts, *Element Types* and relations between them in the *SecLan* conceptual model. For the detailed calculation of α, refer to the literature by Krippendorff [175]. The calculation of α is based on the requirement that reliability data must exhibit some variation [175]. Thus, Krippendorff [175] states that α will be 0 or negative if the matrix is too homogeneous (i.e., most experts agree or disagree on most elements). However, we see it possible that experts may mostly agree or disagree to the concepts, the specific *Element Types* and their respective relations in the *SecLan* conceptual model. This is because we synthesized the *SecLan* conceptual model from data sources systematically (see Section 6.1). Thus, due to the property of Krippendorff's α, we also calculate the *Percentage Agreement PA* [175, 193] (**M3.1.2**), presented in Equation 7.5, and Gwet's Agreement Coefficient (AC) [110] $AC1$ (**M3.1.3**) to provide a complementary view on the agreement between experts.

The *Percentage Agreement PA* calculates the percentage of elements all experts agree upon (N_A) compared to the total number of elements where all experts agree upon (N_A) or disagree upon (N_D) as shown in Equation 7.5.

$$PA = \frac{N_A}{N_A + N_D} \times 100 \tag{7.5}$$

PA allows for a simple assessment of the overall agreement *between* experts [175, 193]. The drawback of this metric is that it does not account for agreements occurring by chance [175, 193]. Gwet's $AC1$ [110] overcomes the drawbacks of both metrics by accounting for agreements occurring by chance and scenarios with high agreement. Furthermore, $AC1$ can handle multiple experts and binary data. For the detailed calculation of $AC1$, refer to the literature by Gwet [110]. The stability of $AC1$ is supported by the study of Wongpakaran et al. [356].

To evaluate whether concepts or relations between concepts are missing in the *SecLan* conceptual model (**Q3.3**) or whether *Element Types* or relations between these *Element Types* in the *System Model* are missing (**Q3.4**), we collect qualitative insights with the metric $Qualitative_{Missing}$ (**M3.3.1**). With this metric we want to gather insights not only about *whether* concepts, *Element Types* or relations are missing but also *which* of them are missing. It is also possible to calculate a metric like the number of missing concepts or relations. However, we see the insights gathered qualitatively as more valuable for judging the completeness of the *SecLan* conceptual model.

With the prior questions, we evaluate whether the *SecLan* conceptual model is *correct* and *complete*. However, applying *SecLan* is not feasible if the *SecLan* conceptual model cannot be applied to represent a broad range of security DSLs and analyzers. This is because then the effort to model security DSLs and analyzers with *SecLan* is too high compared to evaluating the relations manually. Therefore, we ask questions **Q3.5** and **Q3.6** about the capability of the *SecLan* conceptual model to represent existing security DSLs and analyzers:

Q3.5 Is the *SecLan* conceptual model capable of describing existing security DSLs?

Q3.6 Is the *SecLan* conceptual model capable of describing existing analyzers?

We applied *SecLan* to the 66 security DSLs and 33 analyzers providing 408 security checks used in the model synthesis (see Section 6.1). However, as we are the ones who performed the synthesis, evaluating the capability of the *SecLan* conceptual model to represent these security DSLs and analyzers by us would be threatened by bias. Consequently, to answer **Q3.5** and **Q3.6**, we evaluate whether experts agree that the instances of the *SecLan* conceptual model are correct representations of the security DSLs and analyzers respectively. For this purpose, we use *Qualitative Feedback* about the correctness and completeness of the instances of the *SecLan* conceptual model describing security DSLs and analyzers respectively $Qualitative_{Instances}$ (**M3.5.1**).

7.1.2.2. Evaluation Goal 4: Qualitative Exploration of Relations Calculated by SecLan

G4: Qualitatively explore the relations calculated by *SecLan* regarding the correctness of the relationships and further insights regarding the gap between security DSLs and analyzers.

Figure 7.4.: Evaluation goal for evaluating for the quantitative exploration of the relationships calculated by *SecLan*.

The goal of *SecLan* is to provide *Security Checks* of source code analyzers that are capable of detecting non-compliance regarding security-related information captured in *Security Aspects* of security DSLs. Thus, we validate whether *SecLan* is indeed capable of providing such *Security Checks*. For this validation, we perform a qualitative exploration of the relations calculated by *SecLan* through semi-structured expert interviews. We discuss the reasons for the need to apply semi-structured expert interviews in Section 7.2. Semi-structured interviews allow us to obtain valuable insights regarding the relations calculated by *SecLan* and potential shortcomings of our approach, including aspects we do not anticipate a priori. However, the GQM approach does not support explorations where the questions to be answered are not known beforehand. Consequently, we cannot use the GQM approach to guide this exploration. For completeness, we list goal **G4**, although we do not further provide questions or metrics.

7.2. Data Collection Methods

We apply different data collection methods to validate our contributions **C1**, **C2**, and **C3**. In the following, we reason about the suitability of the applied data collection methods with respect to the evaluation goals defined in Section 7.1. Furthermore, we discuss why we cannot feasibly define questions and metrics to evaluate the relations calculated by *SecLan*.

7.2.1. Data Collection Methods for Evaluating Coupling Design and Coupling Approaches

In literature, case studies are one of the most commonly applied data collection methods in software architecture research [170] and to evaluate notations of models for analyses [26]. Thus, applying case studies to investigate evaluation goal **G1** is reasonable because the evaluation examines whether metamodels conform to the reference metamodels, and because our ICs are based on these reference metamodels. Additionally, case studies are the preferred method of related work to evaluate the accuracy of security analyses (e.g., [19, 65, 103, 111, 138, 190, 250, 300, 348, 366]). Thus, we can reasonably use case studies to investigate evaluation goal **G2** as we want to gather data about the accuracy of the coupled analysis. While case studies do not allow for generalized statements, they allow gathering a deeper understanding of the property or the approach under study [282]. We agree with Hahner [111] who refers to the investigated cases *evaluation scenarios* to highlight that the data is not collected in a real-life environment. This is why we adopt this terminology and refer to our cases in the following as *evaluation scenarios*. In Chapter 8, we further define the evaluation scenarios we apply in our study.

7.2.2. Data Collection Methods for Evaluating *SecLan*

For evaluating *SecLan* (**C3**), we use surveys to collect data. We synthesized the *SecLan* conceptual model from reviewing literature. Applying case studies to answer the questions **Q3.1**, **Q3.2**, **Q3.3**, **Q3.4** about the *correctness* and *completeness* of our *SecLan* conceptual model would require to define a ground truth. This ground truth would allow us to determine whether the concepts and relations are complete and correct. Creating this ground truth would require to knowing the actual concepts, relations between the concepts and their semantics in the domains of security DSLs and analyzers. As we do not have this comprehensive knowledge, a potential ground truth would likely also be affected by misinterpretations, resulting in a threat to validity. Because of this threat, Ananieva et al. [15] investigate the correctness and completeness of their conceptual model by *gathering feedback from experts* in the target domains (e.g., by surveys and interviews). This is why we apply surveys to validate whether the *SecLan* conceptual model and the provided *System Model* (*Element Types* and relations) are *complete* and *correct* (**Q3.1**, **Q3.2**, **Q3.3** and **Q3.4**).

We further instantiated the *SecLan* conceptual model for the reviewed 66 security DSLs and 33 source code analyzers to gather quantitative insights into their respective domain and insights into the relationships between *Security Aspects* and *Security Checks*. If these instances are not created correctly, the insights gathered based on them are invalid. The creation of the instances is prone to overfitting and misinterpretations since we created both the *SecLan* conceptual model and instances. Thus, we have to validate whether we *correctly* created the instances of the *SecLan* conceptual model (**Q3.5**, **Q3.6**). Ideally, experts in the domains of the respective security DSLs and analyzers could create the instances themselves to ensure validity. However, we can assume that requesting experts to create

the instances themselves would likely result in limited participation due to the effort required. This challenge becomes evident by the fact that only a small fraction of experts we contacted for our survey was willing to participate (discussed later in Chapter 9). To counteract this challenge, we apply surveys to validate whether we *correctly* created the instances of the *SecLan* conceptual model in our research method (see Section 6.1) because we assume that *reviewing* instances requires less effort than *creating* them.

For validating the correctness of the relationships calculated by *SecLan*, using case studies is similarly challenging as in evaluating the *SecLan* conceptual model. The relationships are calculated by applying regular path expressions we defined in our research. Furthermore, the instances of the *SecLan* conceptual model are also defined by us. We conclude from both aspects that creating a ground truth for these relationships is affected by the same threats to validity as those in evaluating the *SecLan* conceptual model. Therefore, determining whether a relationship is correct requires expert knowledge about the respective security DSLs and security checks. We do not claim to have this knowledge for all security DSLs and analyzers we investigated for synthesizing the *SecLan* conceptual model. Furthermore, the application of *SecLan* to the security DSLs and analyzers resulted in 38,136 relationships between *Security Aspects* and *Security Checks*. This amount makes creating a ground truth infeasible, even if we would have the knowledge about the security DSLs and checks. This is why we apply semi-structured expert interviews to gather feedback about the correctness of the relationships calculated by *SecLan*. Semi-structured expert interviews also allow us to gather additional feedback on unanticipated aspects of the relationships. These additional insights can help us to further validate the relations calculated by *SecLan* and to identify potential shortcomings of our approach. However, the GQM approach requires predetermined questions and metrics to evaluate the relationships calculated by *SecLan*. Because of the unanticipated insights we want to gather, we cannot define these questions and metrics in advance to assess **G4**.

7.3. Summary and Outlook

In this chapter, we have presented the evaluation goals and data collection methods applied to evaluate our contributions. Our evaluation comprises quantitative and qualitative parts. For the quantitative parts, we guide our evaluation by the Goal-Question-Metric (GQM) approach of Basili et al. [23]. Therefore, we present the evaluation goals, the questions we ask to determine whether the goals are achieved, and the metrics we define to investigate whether the questions are answered positively or negatively. We then present the data collection methods we apply to evaluate our contributions and discuss the suitability of these methods for our evaluation goals.

Evaluation Goals, Questions and Metrics The quantitative parts of our evaluation are structured according to the Goal-Question-Metric (GQM) approach of Basili et al. [23]. This approach includes stating the evaluation goals, the questions asked to determine

whether the goals are achieved, and the metrics to investigate whether the questions are answered positively or negatively.

C1 and **C2** (coupling design and coupling approaches) are jointly evaluated. We evaluate the coverage of the reference metamodels and the ICs (**G1**) and the accuracy of the coupled analysis in detecting architectural vulnerabilities that arise from a non-compliant implementation (**G2**). We ask whether the reference metamodels defined in both coupling approaches are covered by the input and output metamodels of the analyses (**Q1.1**, **Q1.2**) and whether the ICs of the specification-based coupling approach are covered in successful couplings on the metamodel level and model level (**Q1.3**, **Q1.4**). To assess the coverage of the reference metamodels, we apply the metric $Coverage_{Ref}$ (**M1.1.1**). Similarly, we apply the metric $Coverage_{IC}$ (**M1.3.1**) to assess the coverage of the ICs. These metrics rely on a generalized *Coverage* calculation that evaluates the coverage of conditions $cond \in COND$ by artifacts A using a function $hold(A, cond)$ to determine if each condition is satisfied. To evaluate the accuracy, we ask how accurate the coupled analysis is in detecting architectural vulnerabilities that arise from a non-compliant implementation. The questions are phrased to investigate architectural vulnerabilities (1) arising from non-compliant information flows in the implementation addressed by the specification-based coupling (**Q2.1**), and (2) from non-compliance due to encryption vulnerabilities in the implementation address by the pattern-search based coupling approach respectively (**Q2.2**). For answering both questions, we use the metrics precision, recall, and F_β-score (**M2.1.1**, **M2.1.2** and **M2.1.3**).

We evaluate **C3** (*SecLan*) regarding the correctness and completeness of the *SecLan* conceptual model (**G3**) and by performing a qualitative exploration to investigate the correctness of the relationships calculated by *SecLan* (**G4**). For evaluating the correctness and completeness of the *SecLan* conceptual model, we ask whether the concepts and relationships between these concepts defined in the *SecLan* conceptual model as well as the specific *Element Types* and relationships between these *Element Types* provided as part of the *System Model* are correct (**Q3.1**, **Q3.2**) and complete (**Q3.3**, **Q3.4**). We rely on expert feedback to investigate this correctness and completeness. To evaluate correctness, we use the inter-rater agreement metrics Krippendorff's α (**M3.1.1**), percentage agreement PA (**M3.1.2**) and Gwet's $AC1$ (**M3.1.3**). These metrics allow us to assess the agreement of all raters regarding the concepts and relationships between the concepts as well as the specific *Element Types* and relationships between these *Element Types*. To evaluate completeness, we define the metric $Qualitative_{Missing}$ (**M3.3.1**) to capture qualitative feedback from experts regarding missing concepts and relationships between the concepts as well as missing *Element Types* and relations between the *Element Types*. Furthermore, we ask whether the *SecLan* conceptual model is capable of describing existing security DSLs (**Q3.5**) and security analyzers (**Q3.6**). We define the metric $Qualitative_{Instances}$ (**M3.5.1**) to capture qualitative feedback from experts regarding the capability of the *SecLan* conceptual model to describe existing security DSLs and security analyzers. For completeness of the GQM plan structure, we also define the evaluation goal **G4** to qualitatively explore the relationships calculated by *SecLan*. However, we do not define questions and metrics for this goal because we want to gather unanticipated insights about the relationships calculated by *SecLan*.

Data Collection Methods For **C1** and **C2** (coupling design and coupling approaches), we use case studies (in the context of this thesis called *evaluation scenarios*), which align with common practice in architecture and security-analysis evaluation and support assessing both coverage and detection accuracy in concrete settings. For **C3** (*SecLan*), we use expert surveys and interviews because creating an independent ground truth for *SecLan* concepts, relations, and instances would be subject to the same interpretation biases as the synthesis itself. The qualitative exploration of relationships calculated by *SecLan* (**G4**) is performed by applying semi-structured expert interviews. Using case studies to evaluate the relationships calculated by *SecLan* is similarly challenging to evaluating the *SecLan* conceptual model because the relationships are calculated by applying regular path expressions we defined in our research and the instances of the *SecLan* conceptual model are also defined by us. Consequently, the same biases and misinterpretations that affect the evaluation of the *SecLan* conceptual model also affect the evaluation of the relationships calculated by *SecLan*. Furthermore, the choice of semi-structured expert interviews is motivated by the large number of relationships (38,136), which makes full survey-based rating infeasible, and by the need to collect additional, unanticipated insights that cannot be captured well with predefined survey questions.

Outlook In the remainder of *Part III - Validation*, we present the study designs and results of our evaluations. Chapter 8 presents the evaluation of the coupling design and coupling approaches. In this evaluation, we provide two study designs to address the individual ground truths necessary for evaluating the pattern-search-based coupling approach and the specification-based coupling approach. The evaluation results are then presented jointly to allow for a better comparison of the two coupling approaches. Chapter 9 presents the evaluation of *SecLan*, where we present the participant selection, the survey design, and interview design. We then present the evaluation results for the survey and the interviews.

8. Evaluation of Coupling Design and Coupling Approaches

Literature: This chapter is based on the following (co-) authored publications: [IEEE TSE 2025], and [IEEE ICSA-C 2025]

In this chapter, we jointly evaluate our coupling design (**C1**) and our coupling approaches (**C2**).

We use an *evaluation scenario*-based design for this evaluation (see Section 7.2). Every evaluation scenario comprises a *coupling scenario* (i.e., an architectural analysis, a source code analysis and a security characteristic to be analyzed for non-compliance) and a *system* to be analyzed. Each system must comprise the following artifacts:

Architectural Description. An architectural description of the system that can be realized with the modeling languages used by the architectural analyses in the coupling scenario.

Implementation. An implementation of the system in a programming language supported by the source code analysis in the applied coupling scenario.

Specification. A specification of values of the security characteristic of the coupling scenario that can be represented with the security DSL used by the architectural analysis. If a specification-based source code analysis is used in the coupling scenario, the specification must also be representable with the specification approach used by the source code analysis.

Although both coupling approaches instantiate the same coupling design (see Chapter 4), they differ in three aspects: (1) the types of security characteristics they investigate (security characteristics of data influenced by information flows and security characteristics describing data encryption), (2) the type of source code analyses they apply (specification-based and pattern-search-based), and (3) the vulnerabilities these analyses detect. These differences require separate study designs that involve selecting the analyses for the evaluation and creating the ground truth (presented in Section 8.4 and Section 8.5).

We present the results of the evaluation jointly in Section 8.6 because both approaches have the same evaluation goals (i.e., coverage of the reference metamodels (**G1**) and accuracy of the coupled analysis (**G2**)), and share the same metrics. Joint result presentation also allows for better comparison of the two coupling approaches.

Before introducing both study designs, we provide details on the coverage metrics for **G1** and details on how we collect the accuracy metrics for **G2**. The coverage metric must

be specialized for the evaluation of specific aspects of the coupling, i.e., the reference metamodels and the ICs (see Section 7.1). We present this specialization in Section 8.1. Furthermore, for calculating the accuracy metrics, the ground truth and a comprehensive interpretation of the analysis result must be provided. Although the ground truth depends on the respective coupling approach, the process we apply to obtain the ground truth and the interpretation of the analysis results are common to both study designs. Consequently, we detail this process and the result classification in Section 8.2. Despite differences in the coupling approaches, both study designs share common requirements and assumptions, which we discuss in Section 8.3.

The evaluation of **C1** and **C2** is complemented with threats to validity of our evaluation in Section 8.8. One important aspect regarding the validity of our results is the reproducibility of our evaluation, which we ensure by providing all artifacts created in the evaluation (e.g., metamodels, models and tooling) in our replication package [270].

8.1. Specializing the Coverage Metrics

We specialize the coverage metric presented in Section 7.1 to evaluate the coverage of the reference metamodels and ICs in the context of our coupling approaches. This specialization includes defining (1) the conditions $COND$ which determine the coverage, (2) the artifacts A to which the conditions are applied, and (3) the function $hold$ to evaluate whether a condition $cond \in COND$ is covered by the artifacts.

8.1.1. Specialization of Coverage Metric for Reference Metamodels

We evaluate the coverage of the reference metamodels for both coupling approaches (**Q1.1** and **Q1.2**) using the coverage metric $Coverage_{Ref}(A_{Ref}, COND_{Ref})$ as described in Equation 8.1. We formulate this metric based on our definition of conformance between reference metamodels and metamodels of analyses, which is independent of specific reference metamodels (Definition 6). Consequently, we can apply the same metric to both coupling approaches even though they provide different reference metamodels.

$$Coverage(A_{Ref}, COND_{Ref}) = \frac{\sum_{cond_{Ref} \in COND_{Ref}} hold_{Ref}(A_{Ref}, cond_{Ref})}{|COND_{Ref}|} \quad (8.1)$$

$$\text{with} \quad hold\colon A_{Ref} \times COND_{Ref} \rightarrow \{0, 1\}$$

$Coverage_{Ref}$ calculates the coverage by utilizing the input or output metamodels of the analysis under study (A_{Ref}). For these metamodels, we consider all metaclasses $a_{Ref} \in A_{Ref}$ and references between those metaclasses $\langle a_{Ref}, a'_{Ref} \rangle$.

Every condition $cond_{r,R} \in COND_{Ref}$ comprises the reference metamodels R and an arbitrary but fixed metaclass r. This condition states that the input or output metamodel of an analysis contains a metaclass conforming to $r \in R$. Definition 6 defines conformance

between a metaclass a_{Ref} in the metamodel of an analysis and a metaclass r in the reference metamodel R. Consequently, we do not capture conditions for the references in the reference metamodels. The reference metamodel for output metamodels is described only for source code analyses. Consequently, we calculate $Coverage_{Ref}$ using conditions $cond_{r,R}$ only for reference metamodel for output metamodels when inspecting source code analyses.

The amount of conditions $cond_{r,R}$ differs between the coupling approaches because the reference metamodels differ in the amount of metaclasses they comprise. For example, the reference metamodels for input metamodels of source code analyses in the pattern-search-based coupling approach only comprise metaclasses for representing *System Elements*. Therefore, only one condition $cond_{r,R}$ is defined for the pattern-search-based coupling approach. In contrast, the reference metamodels in the specification-based coupling approach comprise metaclasses for *System Elements*, *Security Characteristics*, *Annotations*, *Configurations* and *Security Policies*, resulting in five conditions.

For operationalization, the function $hold_{Ref}$ checks for conformance by applying the function $conf$ as specified by Definition 6. As shown in Equation 8.2, we define $hold_{Ref}$ to return 1 if $conf$ returns $true$. Consequently, $hold_{Ref}$ returns 1 if (1) a metaclass a_{Ref} exists in the metamodels of the analyses with a semantically valid mapping to the metaclass r of the reference metamodel captured by $cond_{r,R}$, and (2) for every metaclass r' in the reference metamodel referenced by r, a conforming metaclass a'_{Ref} in the metamodel A_{Ref} exists, which is referenced by the metaclass a_{Ref} from (1). Otherwise, $hold_{Ref}$ returns 0.

$$hold_{Ref}(A_{Ref}, cond_{r,R}) = \begin{cases} 1, & \exists a_{Ref} \in A_{Ref} \colon conf(r, R, a_{Ref}, A_{Ref}) \\ 0, & otherwise \end{cases} \tag{8.2}$$

In summary, $Coverage_{Ref}$ evaluates to 1 (i.e., reference metamodels are fully covered) if $hold_{Ref}$ evaluates to true for all metaclasses of the reference metamodels; thus, all metaclasses of the reference metamodels are covered by the input and output metamodels of an analysis. Otherwise, $Coverage_{Ref}$ evaluates to a value between 0 and 1 depending on how many metaclasses of the reference metamodels are covered by conforming metaclasses in the input and output metamodels of an analysis.

This definition of $Coverage_{Ref}$ allows obtaining insights into how well metamodels of analyses conform to our reference metamodels. Further, our definitions here allow us to calculate *Coverage* for existing analyses to determine whether our coupling approach applies to existing analyses or only to analyses tailored to our coupling. Wherever a reference metamodel cannot be applied to existing analyses, added effort must be expended to adapt the metamodels of the analyses to make the analysis usable with our coupling approach, thus contradicting a black-box coupling.

8.1.2. Specialization of Coverage Metric for Integration Conditions

We evaluate the coverage of the ICs of the specification-based coupling approach (**Q1.3** and **Q1.4**) using the metric $Coverage_{IC}(A_{IC}, COND_{IC})$. With this metric, we evaluate

whether the relations described by the ICs are established in successful couplings by the transformations of our coupling process.

Every IC describes necessary relations between the input and output (meta)models of the analyses to be coupled. Consequently, we define a condition $cond_{IC} \in COND_{IC}$ for each IC, stating that the relations described by the IC exist on the metamodel and model level.

For evaluating these conditions, we must consider the input and output metamodels of the analyses, the metamodel of the *Parameterization Model* and the respective models created in applying the coupling as evaluation artifacts A_{IC}. For evaluating whether a condition $cond_{IC} \in COND_{IC}$ is covered, each transformation generates a correspondence model capturing the correspondences between the elements in the transformations of our coupling process. These correspondence models are used in the operationalization of $hold_{IC}$ to determine whether the relations described by a IC exist on the metamodel and model level. Thus, also the correspondence metamodels and correspondence models are part of the evaluation artifacts A_{IC}. We ourselves define the necessary correspondence metamodels (for all pairs of analyses in the case studies) and the metamodels of the *Parameterization Models* ourselves. To capture correspondences with an implicit configuration in the specification-based coupling (see Subsection 5.1.2), we introduce a metaclass in the correspondence metamodels that is able to reference all metamodels of the analysis. We present the realization of this metaclass in more detail in the replication package [270].

With the evaluation artifacts, we define $hold_{IC}(A_{IC}, cond_{IC})$ based on the classes of ICs, i.e., *correspondence conditions* and *constraint conditions*. We define an IC to be covered if $hold_{IC}$ returns 1 for it. However, only investigating whether metaclasses conform to the ICs (e.g., by investigating whether metaclasses are contained in correspondences in the correspondence metamodels) fails to provide information about whether these metaclasses are actually used in the coupling. Thus, we investigate whether *instances of metaclasses* exist that conform to the IC. This approach is valid because instances and references between instances can only exist if corresponding metaclasses and references between metaclasses are defined.

With this definition, $hold_{IC}$ returns 1 for *correspondence conditions* if (1) for each required correspondence, *at least one* correspondence exists between instances of metaclasses affected by the IC (metamodel level) and (2) correspondences exist between instances of those metaclasses as required by the IC (model level). In our running example, $hold_{IC}(A_{IC}, IC1)$ returns 1 because there is a correspondence between, among others, the instance *high* of the metaclass *Security Level* and the instance *high* of the metaclass *Confidentiality Level* (i.e., the actual target of the coupling scenario) as required on the metamodel and model level by **IC1**.

We define $hold_{IC}$ to return 1 for a *constraint condition* if (1) one instance of at least one or all the metaclasses affected by the IC exists (depending on the condition) that references instances of metaclasses as described by the IC (metamodel level) and (2) at least one or all instances of those metaclasses (depending on the condition) reference instances of metaclasses as described by the condition. In our running example, $hold_{IC}(A_{IC}, IC3)$ evaluates to 1 because (among other reasons) an annotation annotates the *Security Level*

high (**IC1**) to the *Parameter userData* (**IC2**) in the buy service and is captured in an implicit configuration (**IC2**). **IC3** requires the *existence of at least* one such annotation.

In this way, if all ICs are covered on the metamodel and model level, then $Coverage_{IC}$ evaluates to 1 (full coverage). Otherwise, there are ICs that are not covered. This definition of $Coverage_{IC}$ allows obtaining insights into whether the ICs are *necessary*. If $hold_{IC}$ determines that a IC does not hold despite a successful coupling, then the condition is not necessary and $Coverage_{IC}$ returns a value smaller than 1. One consequence is that a *coupling expert* who attempts to enforce a IC which is actually unnecessary for the coupling will likely be hindered by the unnecessary conditions.

8.2. Ground Truth and Result Classification for Evaluating Accuracy

To evaluate the *Accuracy* of the coupling approaches in our evaluation scenarios (**Q2.1** and **Q2.2**), we have to define a ground truth. While the actual ground truths differ for both coupling approaches, we apply the same process to define them.

The ground truth is necessary for calculating the metrics *precision* (**M2.1.1**), *recall* (**M2.1.2**) and the F_β*-score* (**M2.1.3**) for evaluating the *accuracy* of the coupled analyses. Calculating these metrics requires knowledge about the expected results of the architectural analysis in the coupling to classify the reported vulnerabilities (or not reported vulnerabilities, as discussed later) into true positive t_p, false positive f_p or false negative f_n. This definition is equal to both coupling approaches.

8.2.1. Process for Defining the Ground Truth

Creating a ground truth to evaluate the accuracy of our coupling approaches requires the following artifacts and information for each evaluation scenario:

- An architectural system description including a specification of security characteristic for each architectural analysis applied.
- An implementation corresponding to the architectural description.
- In case of a specification-based source code analysis, a specification of the security characteristic for the source code analysis.
- Known vulnerabilities that can be detected with source code analyses applied in our evaluation scenarios, and that can influence the results of the architectural analyses in the coupling scenarios.
- Knowledge about which of the prior vulnerabilities in the implementation would influence the results of the architectural analysis and how.

EC1	**Description:**	*Effect on architectural result*
	Categories:	*effect, no effect*

Table 8.1.: Equivalence class EC1 and its categories for vulnerability injection to create a ground truth

With this information, we have the knowledge about the expected results of the architectural analysis in the coupling for each evaluation scenario. We found no related work that provides such a ground truth for security characteristics of data affected by information flows and security characteristics related to encryption. Therefore, we have to create the comprehensive ground truths ourselves for the evaluation involving the specification-based coupling approach (see Section 5.1) and the pattern-search-based coupling approach (see Section 5.2).

We select case study systems from related work providing either design models with specifications of security characteristics, or implementations with or without known vulnerabilities. We complement these systems with the missing artifacts (e.g., with architectural models if only implementations are provided). These systems are adapted so that they can be used in our evaluation scenarios as ground truth.

For the architectural models, we ensure that an isolated architectural analysis does not report any vulnerabilities. We then obtain knowledge of the results expected from the architectural analysis in the coupling by systematically injecting vulnerabilities into the implementation of evaluation scenario systems. These vulnerabilities are designed so that we obtain values of the security characteristic of data that are non-compliant with the architectural model in the coupling scenario. With this process, the architectural analysis only reports vulnerabilities based on the coupling. Our procedure here relates closely to the evaluation of accuracy in related work [159, 250, 300, 337, 338].

We design the injected vulnerabilities based on equivalence classes (denoted by **EC**$\mathbf{x}$, where $\mathbf{x}$ is a sequentially increasing number). The majority of the equivalence classes are specific to the study design because they are formulated with respect to the vulnerabilities that can be detected by the source code analysis or the capabilities of the coupling approach. We present these equivalence classes in the respective study designs for both coupling approaches (see Subsection 8.4.3 and Subsection 8.5.3).

However, the equivalence class **EC1** (presented in Table 8.1) is common for both coupling approaches. **EC1** describes whether a vulnerability injected into the implementation will have an effect on the architectural analysis result or not. This equivalence class captures the property of our analysis coupling that not every non-compliance between the values specified in the architectural models and the values that result from the implementation leads to an architectural vulnerability (see Subsubsection 4.4.3.3). Non-compliance does not lead to new architectural vulnerabilities if the resulting change of the value of the security characteristic in the coupling scenario due to non-compliance does not violate the security policy. **EC1** contains two categories: *effect* and *no effect*. A vulnerability in the implementation belongs to the category *effect* if the change of a value of the security characteristic in the coupling scenario due to non-compliance will result in a vulnerability reported by the architectural analysis. A vulnerability belongs to the category *no effect*

if the change of a value of the security characteristic in the coupling scenario due to non-compliance will not result in a vulnerability reported by the architectural analysis.

8.2.2. Classification of Architectural Vulnerabilities for the Accuracy Metrics

We now define the classification of a vulnerability reported (or not reported) by the architectural analysis in the coupling into true positive t_p, false positive f_p or false negative f_n. We base this definition on the knowledge about the expected results of the architectural analysis obtained with the process described above. This classification has to consider the categories of equivalence class **EC1** (i.e., *effect* and *no effect*), because they result in different definitions of t_p, f_p, or f_n.

We start by defining the t_p, f_p and f_n for the category *effect* (i.e., vulnerability must arise from non-compliance) as follows. A t_p is a vulnerability that is *reported* by the architectural analysis and is *expected* due to the injected vulnerabilities. A f_p is a vulnerability that is *reported* by the architectural analysis and is *not expected* due to the injected vulnerabilities. This definition of a false positive is conservative. The coupling may have modified values of a security characteristic of data in the architectural model which validly result in a vulnerability, but they were not considered in the study design. Therefore, our results are a lower bound in the metric because a false positive affects the divisor and not the dividend in the metric; thus penalizing potential errors performed by us in the ground truth creation. A f_n is a vulnerability *not reported* by the architectural analysis but is *expected* to be reported based on the injected vulnerabilities.

We define the t_p, f_p and f_n for the category *no effect* (i.e., vulnerability is *not* expected to arise from non-compliance) as follows. A t_p is when *no* vulnerability is *reported* by the architectural analysis as *expected* due to the injected vulnerabilities. Our running example describes a situation where an information flow results in the value of the confidentiality level *high* for the parameter *entry* in the method *log*, but that value is specified in the architectural model as *low*. No vulnerability is reported due to the coupling, because the security policy allows *high* data to be allocated on *Non-EU* resources. Consequently, we would count this outcome as t_p if we inject this information flow deliberately. A f_p is when *no* vulnerability is *reported* by our architectural analysis, although we expect a vulnerability due to the injected vulnerabilities. A f_n is when a vulnerability is *reported* by the architectural analysis, but this vulnerability is *not expected* based on the injected vulnerabilities.

8.3. Requirements and Assumptions for the Study Designs

To evaluate our coupling approaches, we state two requirements that are common to both study designs: that the metamodels and models of the analyses involved in the study design are represented with the EMF [320], and the availability of working tooling. Furthermore,

we make an assumption about the accuracy of the static source code analyses involved in the coupling scenarios.

Requirement: Application of Eclipse Modeling Framework To evaluate the *Coverage* of the reference metamodels and ICs (**G1**), we require the input and output metamodels, and the respective models used in the coupled analyses to be automatically processable (i.e., in a format that can be read and manipulated by tools). Consequently, we require these metamodels and models to be created with the Eclipse Modeling Framework (EMF) [320], a popular open-source framework for model-driven development used in literature [96, 112, 150, 172, 250, 275, 300, 337, 338, 348]. The availability of EMF models is relevant because many design-level analyses use EMF-based metamodels and models as input and output due to its popularity. For all analyses, we use the input and output metamodels, where these exist for EMF; otherwise, we reconstruct the metamodels in EMF to the best of our knowledge. This reconstruction is especially necessary for source code analyses as we found that they commonly do not provide EMF-based metamodels (e.g., [18, 19, 98, 100, 260, 263, 310, 313]) but provide output in formats like the Static Analysis Results Interchange Format (SARIF) [16]. For the input models of analyses, we either reuse them from related work when fully available, adapt available models or recreate missing models manually as described in the sections about the respective study designs (Section 8.4 and Section 8.5). We present adaptations and recreations of the input models in full detail in our replication package [270].

Requirement: Working Tooling For evaluating the *Accuracy* of the coupled analyses (**G2**), we require every analysis to provide *Working Tooling* as we must be able to execute the analyses to obtain the analysis results. We say that tooling is working either if we can download and execute it without further intervention or if we have access to all artifacts and the information on how to build and run it. We discuss the availability of architectural analyses and static source code analyses with *Working Tooling* in the respective sections of our study designs (Subsection 8.4.1 and Subsection 8.5.1).

Assumption: Accuracy of Source Code Analyses In Section 8.2, we discuss the classification of the architectural analysis results after applying the coupling into true positive t_p, false positive f_p, and false negative f_n for our *Accuracy* evaluation. For this classification, we assume that the static source code analysis is accurate (i.e., the analysis detects vulnerabilities if they exist and that it does not raise false alarms). While the accuracy of static information flow analyses is evaluated in the literature [19, 80, 158], the evaluations show that perfect results are hard to achieve and that often over-approximations occur. The same insights are provided in the evaluation of pattern-search-based analyses [65, 103, 138, 190, 366]. This insight is relevant as inaccuracies of the source code analysis in a coupling scenario can also impact the architectural analysis results. The reason is that any information extracted from false alarms in the source code analysis may lead to false alarms in the architectural analysis as well. Also, existing vulnerabilities that are not detected by the source code analyses cannot be used in the architectural analysis. Based on

the inaccuracies found in literature, we admit that this assumption is unrealistic. However, we deliberately assume the accuracy of the source code analyses because of three reasons: First, we can simulate a scenario where a *coupling expert* cannot influence the accuracy of a source code analysis and may not be aware of all details of the selected analysis. Secondly, we can define a ground truth without overfitting to the analysis capabilities; thus, presenting a more realistic evaluation. Lastly, we can investigate the impact of inaccuracies on the results of the architectural analysis.

8.4. Study Design for the Specification-Based Coupling Approach

This section presents our study design involving the *specification-based coupling approach* to collect data for evaluating the coverage of the Reference Metamodels (**Q1.1**), the coverage of the ICs (**Q1.3**, **Q1.4**), and the accuracy achieved by the coupling approach (**Q2.1**).

We create evaluation scenarios by coupling *two* architectural analyses with *two* source code analyses. We select architectural analyses that are able to describe security characteristics of data affected by information flows and source code analyses that use specifications of security characteristics of data to investigate information flows in the implementation. Our selection of four analyses results in *four* metamodels for their input, *four* metamodels for their output, and *two* metamodels for the *Parameterization Model*. The created couplings are applied for *one* security characteristic of data, resulting in a total of *four* coupling scenarios. These *four scenarios* are then applied to *four* systems, resulting in *sixteen evaluation scenarios*. We select systems with different semantics of the security characteristics of data. Two systems specify confidentiality levels with the values *high* and *low*, whereas the other two systems use specifications of roles. Executing our analysis coupling in all 16 evaluation scenarios results in a total of *88* models. We create *eight* times the *Annotated Architecture* (describing four systems for two analyses) instead of sixteen times because the architecture of the system is unaffected by the respective coupled analysis. Furthermore, we create sixteen times the *Annotated Source Code*, *Source Code Analysis Result*, *Parameterization Model*, the modified version of the *Annotated Architecture*, and the *Architectural Analysis Result*. The *Annotated Source Code* is derived from the *Annotated Architecture* for each source code analyses, resulting in a total of *sixteen* versions of the *Annotated Source Code*. The *Source Code Analysis Result* contains the results for each *Annotated Source Code*, resulting in *sixteen* versions of the *Source Code Analysis Result*. The *Parameterization Model* contains the abstraction of the *Annotated Source Code* for each coupling scenario, again resulting in *sixteen* versions of the *Parameterization Model*. The modified version of the *Annotated Architecture* contains the architectural model with the values of the security characteristic modified according to the results of the source code analysis, resulting in *sixteen* versions of the modified *Annotated Architecture*. Finally, the *Architectural Analysis Result* contains the results of the architectural analysis for each modified version of the *Annotated Architecture*, resulting in *sixteen* versions of the *Architectural Analysis Result*.

The presentation of the study design is structured as follows: First, we present the selection of analyses in Subsection 8.4.1. Then the systems used in the evaluation are described in Subsection 8.4.2. Finally, we explain how we create the ground truth for our evaluation in Subsection 8.4.3.

8.4.1. Analysis Selection: Specification-Based Coupling Approach

To select the architectural and source code analyses for our study, we first classify them by specific attributes to achieve a broad set of representative analyses in our evaluation.

We base the attributes on the metaclasses in the reference metamodels as well as on the different representations of those metaclasses in our analyses. Because these attributes are based on the reference metamodels, they are also specific to the specification-based coupling approach.

8.4.1.1. Attributes for Selecting Analyses

We define the following attributes for classifying architectural and source code analyses:

A1: System Abstraction. The analyses in our coupling approach use different systems abstractions to detect vulnerabilities. In this dissertation, we address analyses that use either the *architecture* or the *implementation* to detect vulnerabilities. Attribute **A1** allows to differentiate between architectural and source code analyses. In our case study, we select multiple analyses of the system abstraction to address evaluation threats due to overfitting to one analysis.

- **A2: Values of Security Characteristics of Data.** The values of security characteristics available in an analysis is either *predefined* or *specifiable* (see Section 2.4). One particular case is the use of *no* values of security characteristics of data in source code analyses (e.g., as in FlowDroid [19]). Source code analyses using *no* values detect information flows only between specified sources and sinks. In such analyses, sinks imply system element where information must not flow to, whereas sources imply system elements where information originates from. We cannot use analyses of the value *no* in our case study because the values implied by the sources and sinks differ implicitly based on the semantics of the annotated system elements and the security objective to be analyzed (see paragraph 5.1.2.1). This attribute is important because our ICs influence the security characteristics of data so that any source code analysis with *predefined* values will also constrain the values usable in the architectural specification.
- **A3: Configuration Representation.** Analyses may use an *explicit* configuration or an *implicit* configuration (see paragraph 5.1.2.1). This attribute influences our coupling because the configuration defines which elements of the input models are used to calculate the output. This information is necessary to resolve the information in the *Source Code Analysis Result* to the specifications in the *Annotated Architecture*

Analysis	View	Values	Config. Rep.	Pol. Spec.
Access Analysis [172]	Arch.	Spec.	Implicit	Integr.
DFA [300]	Arch.	Spec.	Implicit	Config.
DFA-Ext	Arch.	Spec.	Implicit	Config.
JOANA [98]	Code	Spec.	Explicit	Config.
CodeQL [100, 214]	Code	Not	Explicit	Config.
CodeQL-Ext	Code	Spec.	Explicit	Config.

Table 8.2.: Selected Analyses for Case Study of specification-based analysis coupling. The values are specifiable (Spec.), predefined (Predef.), or not supported (Not). The configuration is provided *Explicit* by specific elements or *Implicit* by the input models. The security policy is either *Configurable* (Config.) or *Integrated* (Integr.).

in the *Parameterization* step. Consequently, wherever an *implicit* configuration is used by an analysis in a coupling scenario, it becomes necessary to realize definitions for exactly which analysis in the coupling uses or provide exactly which in- or output model. In our evaluation, we address this need for identification by the configurations in our coupling tool for performing the coupling process. However, for our evaluation of the *Coverage* of the ICs (**Q1.3**, **Q1.4**), we must be able to capture these correspondences in the correspondence metamodels (see Section 8.3).

- **A4: Policy Specification.** The security policy is either *configurable* by the analysis user or it is *integrated* into the analysis (see Section 2.4). For the values of the security characteristics to be mapped correctly and for the validity of the ICs to be checked, *integrated* policies must be recreated by the coupling expert as they are encoded in the analysis techniques. This is important to avoid using different policies in both of the coupled analyses regarding the same security characteristics. Recreating an *integrated* policy requires knowledge about the analysis algorithm. Errors in recreating an *integrated* policy directly influence our coupling because the source code analysis may report vulnerabilities that do not fit the *security policy* used by the architectural analysis.
- **A5: Information Flow Domain.** In the specification-based coupling we focus on security characteristics of data that are influenced by the information flows in the system. Thus, we have to select architectural analyses which comprise specifications of security characteristics of data which are influenced by the information flows in the system. Furthermore, we have to select source code analyses that are able to detect non-compliant information flows in the implementation based on specifications of security characteristics of data.

Table 8.2 shows this classification of analyses by attribute. We omit the attribute *Information Flow Domain* because it is a mandatory attribute in every analysis of our study design. We selected analyses that cover different combinations of the other attributes to evaluate our coupling approach in a broad manner. Furthermore, we selected analyses with working tooling to enable an automated evaluation (see Section 8.3). However, we only found few analyses with working tooling. We discuss the selected analyses in the following.

8.4.1.2. Architectural Analyses Selected for the Specification-based Coupling Approach

We use the Access Analysis [172] and an extended version of the DFA [300] as architectural analysis (**A1**). Access Analysis provides *DataSets* which can be attached to parameters of services (security characteristics of data). The values for these *DataSets* are *specifiable* (**A2**). The values specify expectations about internal information flows which are used to calculate whether adversaries can gain unallowed access to data. For this purpose Access Analysis defines adversaries, the locations they can enter, protection mechanisms they can circumvent and the data they are allowed to access. All information for performing the analysis is provided in the input models (*implicit configuration*; **A3**). The security policy of the Access Analysis is *integrated* (**A4**). Whether an adversary can gain unallowed access is evaluated based on Prolog clauses that are part of the analysis implementation. These Prolog clauses use, amongst others, the *DataSets* assigned to parameters of services to determine whether an adversary can access data allocated to those parameters which they are not allowed to obtain.

We use an extended version of DFA [300] as architectural analysis as well. Our reason for extending DFA (henceforth, DFA-Ext) is that we did not find an architectural analysis that has working tooling, covers our attributes **A2** (Values of security characteristics of data) and **A4** (Policy Specification), and that also annotates architectural elements that have corresponding elements in the implementation (e.g., parameters). This extension of DFA is made possible because the condition for invalid flows (i.e., what is reported by the analysis) is configurable in the analyzer of the DFA [34]. This allows us to add an *Annotation* to parameters in services of components, thus enabling correspondences to the implementation. We use this annotation to provide two checks in addition to the checks described for DFA [300]. The first check investigates whether components are allowed to be deployed on resources based on the values specified for the parameters of the services provided by a component. This check is similar to the analysis of the Access Analysis. The second check investigates whether the value of the security characteristic of data provided by users into the system conforms to the value specified for the parameter of the services receiving this data. This check is similar to a taint analysis (e.g., [98, 224]). The extensions of DFA is further detailed in the replication package [270]. Our DFA-Ext thus conforms to the architectural analysis in our running example. DFA-Ext is a valid analysis because we maintain the basic analysis capabilities of DFA. DFA-Ext uses *specifiable* value ranges (**A2**). Furthermore, DFA-Ext uses no specific configuration element besides the input models (i.e., *implicit* configuration (**A3**)). The security policy of the DFA-Ext is configurable by (1) defining unallowed relations between values of security characteristics specified for resources, and (2) defining unallowed data flows based on the values of security characteristics of data (**A4**).

8.4.1.3. Source Code Analyses Selected for the Specification-based Coupling Approach

Although numerous source code information flow analyses are available in literature (e.g., [19, 37, 98, 100, 105, 224]), only a few of them have readily available, automated, and

working tooling. In addition, we found that some source code analyses with working tooling do not fit our reference metamodels properly. The reason is that analyses like CodeQL [100, 214] or FlowDroid [19] are designed to detect information flows but do not utilize security characteristics for this purpose. To address the issue of only a few source code analyses having working tooling, we extend such analyses if we see the potential for extending their input metamodels, output metamodels, and the functionality to consider security characteristics of data in calculating information flows. Therefore, we select the source code (**A1**) information flow analysis *JOANA* [98] and CodeQL [100, 214] for our case study because they provide working tooling.

JOANA is applied in related work [159] and uses *specifiable* value ranges (**A2**). To distinguish different executions of the analysis, *JOANA* uses *explicit* configurations (**A3**), called EntryPoints. The security policy of *JOANA* is *configurable* because it uses lattices defined in each EntryPoint (**A4**) which require defining allowed flows between the values of the security characteristic of data. *JOANA* is limited to the programming language Java. As *JOANA* does not provide (EMF) metamodels for its input and output, we created these metamodels and the metamodel for the *Parameterization Model* ourselves (see Section 8.3). For this creation, we first provide a rudimentary *Java* metamodel in EMF covering packages, classes, interfaces, methods, parameters and fields. There are existing Java metamodels in EMF such as JaMoPP [119]. However, these metamodels cover the whole Java language whereas we only require a small subset of Java for our case study. Thus, we created a simplified Java metamodel ourselves to reduce complexity. We then analyzed the input and output of *JOANA* to create the respective metamodels in EMF. Here, we provide a metamodel that captures *EntryPoint*, *Lattices*, *Security Level* and *Annotations* of *Security Level* to *Parameter* or *Field* of the Java metamodel. For the output metamodel, we provide a metamodel replicating the structure and information in the *syntax* of the output of *JOANA*. For example, *JOANA* reports parameters in the implementation by the fully qualified name of the class, the name of the method, the type of the parameter and the index of the parameter in the method signature. Thus, we create a metaclass *Parameter* in the output metamodel with the respective attributes. As we will later discuss in our evaluation results (Section 8.6), we found that the output of *JOANA* does not fully conform to our reference metamodels. However, we still use *JOANA* in our case study because we can provide a semantically equal EMF metamodel which conforms to our reference metamodels. We describe all metamodels in detail in the replication package [270].

CodeQL [100, 214] detects flows between sources and sinks. It allows the analysis user to define a *FlowState* for the information flow analysis, which is similar to the security characteristic of data. Unfortunately, no comprehensive description of the application of *FlowStates* exists for the programming language Java, which we are limited to because of *JOANA*. We also contacted CodeQL developers over their GitHub forum to clarify how *FlowStates* can be applied in Java analyses. However, also here no comprehensive description could be provided. Consequently, we classify the CodeQL information flow analysis with the value *No* (**A2**). This is why we cannot apply CodeQL to investigating information flows based on security characteristics of data. However, CodeQL's input, the question asked, and output are modifiable. Therefore, guided by our reference metamodels, we noninvasively extend CodeQL (henceforth, *CodeQL-Ext*) to detect unallowed information

flows based on security characteristics of data. We created EMF [320] metamodels for its input and output.

Our *CodeQL-Ext* introduces metaclasses to define security characteristics of data (*Security Characteristic*), to annotate them to parameters of Java methods and to fields in Java classes (*Annotation*), and lastly to define an information flow lattice (*Security Policy*). For the representation of Java methods and fields, we use the same simplified Java metamodel as for *JOANA*. Also, we extend the structure of reported illegal information flows (*Result Entry*) to tuples of implementation elements and security characteristics of data (*Result Entry Element*). Accordingly, the source code analysis (**A1**) *CodeQL-Ext* utilizes *specifiable* value ranges for security characteristics of data (**A2**). Like CodeQL, our *CodeQL-Ext* has an *explicit* configuration (**A3**) because each query file has an identifier. The security policies of *CodeQL-Ext* are *configurable* (**A4**) because they are given as functions in the CodeQL query. For more details about the metamodels and the extension of CodeQL, please see the replication package [270].

8.4.2. System Selection: Specification-Based Coupling Approach

For evaluating the specification-based coupling approach, we require systems for which all of the following is present: architectural models, the implementation in form of source code, specification of security characteristics of data influenced by information flows, information flows in the implementation which violating the specification and that affect the architectural security. We did not find many publicly available systems that meet these requirements.

Additionally, *JOANA* [98] and CodeQL [100, 214] are whole-program analyses that calculate the information flows for monolithic programs with call-return semantics. This is why Katkalov et al. [159] generate source code in a single code base for the information flow analysis with *JOANA*, although they design message-oriented applications. Consequently, the system implementations are required to use call-return semantics. To avoid further reduction due to this requirement, we reimplement selected message-oriented or distributed systems as monolithic programs with call-return semantics. For a full description of all architectural models and implementations, and for the actual architectural models and implementations see the replication package [270].

The systems we selected are *TravelPlanner* [159, 211, 300], *JPMail* [300, 338], *EclipseSecureStorage* [76, 250, 337], and CoCoME [127]. TravelPlanner and CoCoME use security characteristics of data with the semantics of roles. JPMail and EclipseSecureStorage use security characteristics of data with the semantics of confidentiality levels. Figure 8.1 shows excerpts of the information flow policies of TravelPlanner (1) and JPMail (2), where each policy comprises the values of the respective security characteristics of data of our coupling scenario and the allowed flows between them.

Distinctive semantics in the values of the security characteristics of data allows us to counteract threats that the coupling approach can only handle a specific type of semantics for security characteristics of data. In two of our case study systems (TravelPlanner

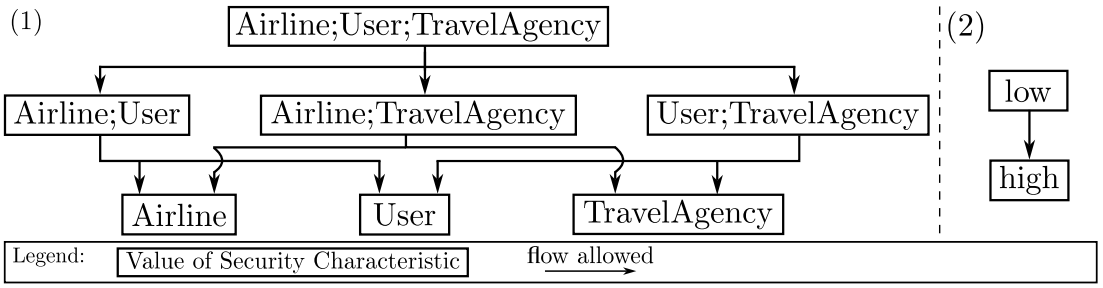


Figure 8.1.: Examples of applied classes of case study information flow lattices for (1) TravelPlanner (reduced set of values) and (2) JPMail.

and CoCoME), the security characteristic of data represents a *role* in a role-based access scenario with *disjunctive values*. In the other two systems (EclipseSecureStorage and JPMail), a security characteristics of data represents the *confidentiality level*. Roles in scenarios as opposed to confidentiality levels impose different forms of values and lattices as illustrated in Figure 8.1. The value of a role may be a combination of basic values, but the value of a confidentiality level may not (see Subsubsection 2.3.2.2). Disjunctive values in a lattice may provide multiple paths between values, but with confidentiality levels, only one single path leads through the lattice. We had to create different transformations to address these distinctive semantics. (For implementations of all transformations, see our replication package [270])

TravelPlanner TravelPlanner enables users to book flights. To describe the roles an adversary must fulfill to be allowed access to specific data, the TravelPlanner specification [159, 211] assigns role-based values like *TravelAgency*, *User*, and *Airline* to data (security characteristic of data). To establish the ground truth (described in Section 8.2), we add the role *Support* (omitted from Figure 8.1 for simplicity). For all roles and their combination, TravelPlanner uses a *disjunctive values*. TravelPlanner is used in related work to evaluate IFlow [159], DFA [300], and Access Analysis [172]. This makes TravelPlanner relevant to our evaluation. The architectural models of TravelPlanner already exist for DFA [300] and Access Analysis [172], so we can use these in the evaluation. Further, the IFlow approach [159] is also relevant to our coupling because IFlow enables analysis of non-compliance between the designed system and the implementation.

CoCoME *CoCoME* [127] is a community case study for component-based systems which describes a trading system in supermarkets that manage sales using cash desks. CoCoME was extended by Greiner and Herda [104] — calling it *CoCoME with Security* — to inspect information flow security. They describe the system, adversaries in the system and security characteristics of data in the form of roles like *BuyStander*, *Customer*, and *Cashier*. CoCoME with Security describe *disjunctive values* for roles. Related work use CoCoME or CoCoME with Security to evaluate information flow analyses [96, 105] or compliance analysis [337]. Therefore, CoCoME and CoCoME with Security are relevant for our evaluation. Greiner and Herda [104] also provide a model of CoCoME with Security for the Access Analysis

metamodel. However, this model lacks elements to describe, for example, adversaries and deployment. We extend CoCoME with Security (as described by Greiner and Herda [104]) using information from the original CoCoME description [127] for our evaluation. In the design and implementation of this extension, we find that Access Analysis and DFA-Ext report vulnerabilities. However, our accuracy evaluation requires a system design that is free of vulnerabilities prior to coupling (detailed in Section 8.2). Therefore, to establish an initial system without any security vulnerabilities, we modify the architectural models and the implementation (please see the replication package [270] for all details). Henceforth, we refer to CoCoME with Security simply as CoCoME.

JPMail *JPMail* is a message-oriented email exchange system. JPMail is used in related work to evaluate SecDFD [337, 338] and DFA [300]. Thus, JPMail is also relevant for our evaluation. JPMail uses confidentiality levels as the security characteristics of data with the values *high* and *low* and uses a sequentially ordered lattice. Also, Tuma et al. [338] specify the security characteristic *Attack Zone* and data encryption. We reuse the architectural models of JPMail by the DFA [300] and adapt it for the Access Analysis and the DFA-Ext.

EclipseSecureStorage *EclipseSecureStorage* [76] enables the storage of passwords by Eclipse Plugins and also the recovery of lost master passwords via a challenge response procedure. EclipseSecureStorage is used by Peldszus et al. [256] to evaluate an approach for assessing compliance between designed and implemented information flows. This makes EclipseSecureStorage also relevant to our evaluation. Similar to JPMail, Peldszus et al. [256] uses confidentiality levels with the values *high* and *low* (security characteristics of data) to specify a sequentially ordered lattice for EclipseSecureStorage. EclipseSecureStorage is designed as a single application. This is reflected in the setup process for recovering the master password. Here, the challenges are communicated unencrypted while the responses are communicated encrypted. However, the DFA-Ext and Access Analysis use a component-based architectural design. There, the components are allocated on different resources and communicate through communication links. Consequently, we have to model the EclipseSecureStorage as component-based architecture. To that end, we simulate for EclipseSecureStorage an evolution step where the management logic and the storage of information are performed by separate components located on different resources. When applying DFA-Ext and Access Analysis to ensure that the new design is free of vulnerabilities (similar to *CoCoME*), DFA-Ext reported a vulnerability because the questions are sent unencrypted. We adapt the design and implementation so that the DFA-Ext does not report any vulnerabilities (as detailed in the replication package [270]).

8.4.3. Ground Truth: Specification-based Coupling

We now describe the creation of our ground truth by systematically injecting information flows into the implementation of the systems. For these information flows we also define the architectural results we expect to be reported by the source code analyses. We follow

EC2	**Description:**	*Values of security characteristics of data flowing to an implementation element*
	Categories:	*same, different*
EC3	**Description:**	*Visibility of sources of an information flow*
	Categories:	*visible, hidden*

Table 8.3.: Equivalence classes EC2 and EC3 and their categories used for vulnerability injection to create a ground truth for the specification-based coupling approach

the process for defining the ground truth as described in Subsection 8.2.1, by describing the equivalence classes we use to guide the injection of information flows into the implementation.

8.4.3.1. Equivalence Classes for Injecting Information Flows

The injection of information flows is guided by three equivalence classes (**EC1**, **EC2**, and **EC3**). These equivalence classes are intended to influence the coupling and the result of the architectural analysis. We use **EC1** as defined in Subsection 8.2.1. **EC2** and **EC3** are defined in the following and are summarized in Table 8.3. We inject information flows so that each equivalence class is covered at least once in at least one of the evaluation scenarios. Through the injected information flows, we demonstrate the capabilities and shortcomings of our coupling approach.

Equivalence Class EC2 The accuracy of the coupling highly depends on the calculation of the values of security characteristics resulting from the implementation in the *Parameterization* step. The complexity of this calculation increases wherever it becomes necessary to consider multiple non-compliant information flows of different values of a security characteristic flowing to an implementation element. The calculation of the value resulting from the implementation then requires exact knowledge about the semantic meaning of the values and also about the relation between the combined values. If the coupling expert misinterprets the semantics of the combination of values as well as the effect of these values on the architectural analysis, then the coupling creates false results. Therefore, we introduce the equivalence class **EC2** with the categories *same* and *different* as summarized in Table 8.3. The category *same* introduces one flow or multiple flows from implementation elements with the same values of a security characteristic of data to a single implementation element; as a result, the architectural specification is violated by the identical value of the origins of the flows. The category *different*, on the other hand, introduces multiple flows from implementation elements with different values of a security characteristic of data to a single implementation element; as a result, the architectural specification is violated by the value resulting from the combination of the different values from the origins of the flows.

System	Semantics	Arch. Analysis	# Vul.
TP	roles	Access Analysis	7
TP	roles	DFA-Ext	4
JP	confid.	Access Analysis	8
JP	confid.	DFA-Ext	4
CCM	roles	Access Analysis	3
CCM	roles	DFA-Ext	2
ESS	confid.	Access Analysis	1
ESS	confid.	DFA-Ext	1
Total			30

Table 8.4.: The Systems, the semantics roles (*roles*) or Confidentiality Levels (*confid.*) for the security characteristics of data, the architectural analysis and the number of expected architectural vulnerabilities (# Vul.) after the coupling. Systems are denoted with JP (JPMail), TP (TravelPlanner), CCM (CoCoME) and ESS (EclipseSecureStorage).

Equivalence Class EC3 Our running example also illustrates how the coupling allows detecting architectural vulnerabilities caused by information flows from implementation elements without correspondence to the architectural models (e.g., from fields in classes). The third equivalence class **EC3** captures the impact on architectural analysis results of information flows originating from implementation elements which cannot be represented in the architectural model. **EC3** contains the categories *visible* and *hidden*. The architectural analyses use the PCM, which contains elements corresponding to parameters *but not to fields* in the implementation. The category *visible* comprises information flows originating from parameters of methods, because parameters of methods can be mapped to the PCM. The category *hidden* comprises information flows originating from fields of classes, because these cannot be mapped to the PCM. **EC3** allows investigating the impact of the coupling so that we are able to detect architectural vulnerabilities due to information flows in the implementation originating from elements not visible in the architectural model.

8.4.3.2. Injected Information Flows

We inject seven information flows into TravelPlanner, six into JPMail, one into CoCoME, and one into EclipseSecureStorage based on **EC1**, **EC2**, and **EC3**. The injected information flows are realized either by omitting declassification mechanisms (e.g., encryption) or by establishing an information flow that contradicts the given policy. Listing 8.1 illustrates the changes in the implementation of the service *setupRecovery* of *EclipseSecureStorage*. Here, the injection of an information flow into *internalPut* is realized by removing the prior encryption of data. A similar omission of declassification mechanisms to evaluate accuracy exists in related work (e.g., [112, 159, 300, 338]). *At least one* injected information flow of this type is found in each of our case study systems. In Table 8.4, we list the number of expected architectural vulnerabilities reported after applying the coupling. Based on

```
setupRecovery(String[][] challengeResponse, moduleID, container) {
   root = container.getRootData();
   node = recoveryNode(root, moduleID);

   internalPwd = mashPassword(challengeResponse[1]);

(-)internalPasswordExt = new PasswordExt(
(-)     new PBEKeySpec(internalPwd.toCharArray()),RECOVERY_PSEUDO_ID);
   password = root.getPassword(moduleID, container, false);

(-)data = StorageUtils.getBytes(new String(password.getPassword()));
(-)encryptedValue = root.getCipher().encrypt(internalPasswordExt, data);
(-)node.internalPut(PASSWORD_RECOVERY_KEY, encryptedValue.toString());
(+)data = new String(password.getPassword());
(+)node.internalPut(PASSWORD_RECOVERY_KEY, data + internalPassword);
   root.markModified();
    //additional code...
}
```

Listing 8.1: Example of an injected information flow illustrated on a reduced version of the setupRecovery method of EclipseSecureStorage. Red text and (-) indicates removed code and blue text and (+) indicates added code. Most data types and error-handling omitted to improve readability.

the injected information flows, we expect the following number of vulnerabilities after applying our coupling approach:

- For *TravelPlanner*, we expect Access Analysis to detect *seven* vulnerabilities and DFA-Ext to detect *four* vulnerabilities. We injected information flows of the category *no effect* for which we expect the integration of *one* value of security characteristic of data. For this value, Access Analysis and DFA-Ext should not report a vulnerability.
- For *JPMail*, we expect Access Analysis to detect *eight* vulnerabilities and DFA-Ext to detect *four* vulnerabilities.
- For *CoCoME*, we expect Access Analysis to detect *three* vulnerabilities and DFA-Ext to detect *two* vulnerabilities.
- For *EclipseSecureStorage*, we expect Access Analysis and DFA-Ext to detect *one* vulnerability.

To facilitate understanding *how* the injection is realized, we describe examples of the injected flows and the expected results of the architectural analyses after the coupling. However, detailing every injected flow and the corresponding expected results for each architectural analysis would not improve the understanding of our study design in this section. For details about *all* injected information flows, please refer to the replication package [270] for details.

TravelPlanner In TravelPlanner, we inject an information flow described by Katkalov et al. [159] but also investigated in the evaluation of the DFA [300]. This information

flow originates (without a declassification) from the *hidden* field *ccd* (**EC3**) of the *Class CreditCardCenter* with the value of the security characteristic of data *User* to the parameter *ccd_decl* of the *Method bookFlight* in the class *Airline* with the value of security characteristic of data *Airline*. Since only one information flow to the sink parameter exists, the *same* (**EC1**) values flow to the parameter *ccd_decl*. This information flow must have an *effect* (**EC2**) in the architectural analyses because it transports data intended only for the user to a system element visible to the *Airline*. We introduce another information flow with the *same* values of security characteristics of data (**EC1**) from the *hidden* field *passengerDetails* (**EC3**) of the TravelPlanner class with the value of the security characteristic of data *User* to a parameter *details* of the method *addToFlight* in the FlightStoring class specified with the value *Airline*. There must be *no effect* (**EC2**) in DFA-Ext because the resource containing the corresponding *FlightStoring* component that provides the method *addToFlight* is not accessible by any actor. In the calculation of vulnerabilities, Access Analysis considers required and provided interfaces of components, but DFA-Ext considers only provided interfaces of components. Therefore, in Access Analysis, there must be *no effect* (**EC2**) regarding the provided interface in the FlightStoring class, but conversely, there must *be* an *effect* (**EC2**) in the required interface of the Airline class.

Due to the injected information flows, we expect Access Analysis to detect *seven* vulnerabilities and DFA-Ext to detect *four* vulnerabilities in TravelPlanner after we apply our coupling approach. Due to the injected information flows of the category *no effect*, we expect the integration of *one* value of security characteristic of data, which must not be reported either by Access Analysis or by DFA-Ext.

JPMail In JPMail, we inject two information flows by removing the declassification of the *header* and *body* of an email, as performed already in the evaluation of DFA [300]. For the classification of our injected flows, we use the specification of Tuma et al. [338] and Seifermann et al. [300] of *Attack Zones*. One injected information flow originates from the *visible* parameter *header* in the method *sendEmail* of the class MailSendingFacade (**EC3**) with the value of security characteristic of data *high* to the parameter *header* of the method *dispatch* in the class POP3 with the value *low*. Because there is only one path through the system, this injected flow leaks the *same* value of security characteristic of data (**EC1**) to the *header* in the dispatch method. The flow also has an *effect* (**EC2**) because the POP3 component is located in an *Attack Zone*. Due to these injected information flows, we expect Access Analysis to detect *eight* vulnerabilities and DFA-Ext to detect *four* vulnerabilities in JPMail after we apply our coupling approach. Again, because there is only one path through the system, we cannot inject an information flow with the category *no effect*, even when we change the value of a security characteristic of data due to the coupling into the architectural model.

Due to the injected information flows, we expect Access Analysis to detect *eight* vulnerabilities and DFA-Ext to detect *four* vulnerabilities in JPMail after we apply our coupling approach.

CoCoME In CoCoME, we inject a transitive information flow from the *visible* (**EC3**) parameter *number* (classified with the roles *Customer* and *Bank*) in the method *readCardNumber* of the class *CardReader* to the parameter *number* in the method *readCardNumber* of the class *CashDesk*. In this injected flow, the declassification of the *number* parameter is omitted. The class *CashDesk* forwards the content of the *number* in the method *readCardNumber* to the parameter *cardNumber* of the method *printPaymentCard* of the class *Printer*. Thus, this transitive injected flow has an *effect* (**EC2**) because the parameter *number* of the *readCardNumber* method and the parameter *cardNumber* in the *printPaymentCard* method are both specified to contain data for the roles *Customer*, *ByStander*, *Bank* and *StoreManager* and *Cashier*. This specification contrasts the values *Customer* and *Bank*, which are specified for the source of the information flow. Because we insert a single information flow, this flow leaks the *same* (**EC1**) security characteristic of data values.

Due to the injected information flows, we expect Access Analysis to detect *three* vulnerabilities and DFA-Ext to detect *two* vulnerabilities in CoCoME after we apply our coupling approach.

EclipseSecureStorage In EclipseSecureStorage, we inject a single information flow from the *visible* (**EC3**) parameter *challengeResponse* of the method *setupRecovery* provided by the class *PasswordManagement*, classified with the value of security characteristic of data *high*. This information flow has already been investigated by Peldszus et al. [256]. As illustrated in Listing 8.1, we remove the encryption of the password derived from the answers in *challengeResponse* to provide, instead, the password directly to the method *internalPut* of the *SecurePreferences* component by the parameter *value* classified as *low*. In contrast, we do not remove the encryption of the questions in EclipseSecureStorage described in Subsection 8.4.2 since we are the ones who introduced it to avoid encountering an architectural vulnerability. Removing the encryption of the question would render the reported vulnerability unusable in our evaluation, because we already know that retaining the encryption will cause a vulnerability. Since we inject just a single flow, it leaks the *same* security characteristic of data (**EC1**), from the *high* classified *challengeResponse* of the method *setupRecovery* to the *low* classified *value* of the method *internalPut*. This injected flow has an *effect* (**EC2**) because we model *SecurePreferences* and its methods to be accessible by an *Attacker*.

Due to the injected information flow, we expect Access Analysis and DFA-Ext to detect *one* vulnerability in EclipseSecureStorage after we apply our coupling approach.

8.5. Study Design for the Pattern-Search-Based Coupling Approach

This section presents the study design involving the *pattern-search-based coupling approach* to collect data for evaluating the coverage of the reference metamodels (**Q1.2**) and the accuracy achieved by the coupling approach (**Q2.2**). In Section 5.2, we describe challenges that

can arise in the pattern-search-based coupling approach in its current designe. Therefore, the study design for the pattern-search-based coupling does not only focus on answering the questions **Q1.2** and **Q2.2**, but also on illustrating these challenges.

We create evaluation scenarios by coupling *one* architectural analysis with *three* source code analyses indivudiually. We select architectural analyses that use security characteristics which specify the encryption of data and source code analyses that detect patterns indicating vulnerabilities that affect data encryption. Our selection of four analyses results in *four* metamodels for their input, *four* metamodels for their output, and *three* metamodels for the *Parameterization Model*. The couplings are applied for *one* security characteristic of data, resulting in a total of *three* coupling scenarios. These *three scenarios* are then applied to *three* systems, resulting in *nine* evaluation scenarios. Executing our analysis coupling in all nine evaluation scenarios results in a total of *48* models. We describe *three* times the *Annotated Architecture*. Furthermore, we create *nine* times the *Annotated Source Code*, the *Source Code Analysis Result*, the *Parameterization Model*, the modified version of the *Annotated Architecture*, and *Architectural Analysis Result*. The *Annotated Source Code* is derived from the *Annotated Architecture* for each source code analyses, resulting in a total of *nine* versions of the *Annotated Source Code*. The *Source Code Analysis Result* contains the results for each *Annotated Source Code*, resulting in *nine* versions of the *Source Code Analysis Result*. The *Parameterization Model* contains the abstraction of the *Annotated Source Code* for each evaluation scenario, again resulting in *nine* versions of the *Parameterization Model*. The modified version of the *Annotated Architecture* contains the architectural model with the values of the security characteristic modified according to the results of the source code analysis, resulting in *nine* versions of the modified *Annotated Architecture*. Finally, the *Architectural Analysis Result* contains the results of the architectural analysis for each modified version of the *Annotated Architecture*, resulting in *nine* versions of the *Architectural Analysis Result*.

The presentation of the study design is structured as follows: First we present the selection of analyses in Subsection 8.5.1. Then the systems used in the evaluation are described in Subsection 8.5.2. Finally, we explain how we create the ground truth for our evaluation in Subsection 8.5.3. An essential part of our study design is applying the *Security Relations* metamodel and the coupling model, which is described in Subsection 8.5.4. Subsection 8.5.4 diverges from the study design for the specification-based coupling approach because the latter does not require techniques to calculate the semantic relations between the models when performing the coupling.

8.5.1. Analysis Selection: Pattern-Search-Based Coupling Approach

We select architectural analyses specifying security characteristics requiring *encryption* and source code analyses finding patterns indicating vulnerabilities that invalidate the *encryption*. In contrast to the specification-based coupling approach, we cannot define a set of attributes regarding our coupling approach for the selection of analyses which hold for the architectural analyses *and* source code analyses. We cannot define these attributes because the analyses do not use the same underlying concept to detect vulnerabilities:

While architectural analyses detect vulnerabilities by investigating whether the security characteristics specified in the architectural model violate a security, pattern-search-based source code analyses find patterns indicating vulnerabilities that affect the security of the system. This contrasts the specification-based coupling approach, where the architectural analyses and source code analyses both detect vulnerabilities based on security characteristics of data specified in the input models. As a consequence, we define separate selection criteria for architectural analyses and source code analyses.

8.5.1.1. Architectural Analyses Selected for the Pattern-search-based Coupling Approach

Our pattern-search-based coupling approach is described for architectural analyses that specify security characteristics requiring *encryption* for architectural elements representing communication. Furthermore, the pattern-search-based coupling approach is described for component-based architectural models where we *do not* consider behavioral information (see Section 5.2). Thus, the capability to specify security characteristics requiring *encryption* on structural architectural elements representing communication, and *working tooling* (see Section 8.3) are our main selection criteria for architectural analyses. While there are several design analyses available that use some form of encryption (e.g., [12, 154, 172, 300, 338]), many of them do not fit to our selection criteria. For example, some analyses require behavioral information [300, 338], whereas for others we could not find working tooling (e.g., for the approach of Almorsy et al. [12]). Thus, already the selection of architectural analyses shows the limitation of omitting behavioral information in architectural models for the coupling, discussed in Section 4.5.

We see this limitation as valid and, therefore, only illustrate the feasibility of our pattern-search-based coupling approach in this evaluation. We discuss future work to extend our coupling design and pattern-search-based coupling approach to behavioral models in Section 11.3.

For demonstrating the feasibility of our coupling approach, we select Access Analysis [172] already applied in the study design of the specification-based coupling approach (see Subsection 8.4.1). Access Analysis annotates communication links of the PCM [275] with the security characteristic *encryption*, indicating that the data which is communicated over this link is expected to be encrypted. A vulnerability is reported by Access Analysis if an *Adversary* can access a communication link that transmits unencrypted data the adversary is not allowed to obtain. In contrast to the study design of the specification-based coupling approach, the security characteristic *encryption* for communication links is relevant for our evaluation because it specifies that communication is designed to be encrypted.

Source Code Analyses We found many source code analyses that detect patterns indicating weaknesses invalidating *encryption*. However, these analyses vary only limited in the patterns they detect and how they present their findings. Because of this limited variation, performing a coupling for all pattern-search-based source code analyses does not provide additional insights at some point.

Therefore, we select a limited set of source code analyses that cover all the following attributes:

- **A1: Encryption Domain.** The analyses find patterns related to weaknesses invalidating *encryption* in source code. There are different types of weaknesses that invalidate *encryption*.
- **A2: Free-Of-Use.** There is a large variaty of pattern-search-based analyses to be selected from. However, many of them are provided as commercial applications. To facilitate reproducibility of our results and to allow broad applicability of our coupling approach, the analyses must be *free-of-use*. There are free versions of commercial tools that provide a limited set of features, but these features are sufficient for our evaluation.
- **A3: Machine-Readable Results.** We found that some *free-of-use versions* of commercial analyses only provide a graphical user interface to display the results. , However, the analyses must provide *machine-readable results* (e.g., as JSON or XML) to facilitate automated coupling. Therefore, we only select analyses that provide machine-readable results.
- **A4: Programming Language Java** The implementations of our evaluation systems are written in Java. Thus, the analyses must support Java as programming language. Selecting analyses supporting Java allows us to focus on the coupling approach instead of dealing with additional complexity introduced by multiple programming languages. Furthermore, Java is still one of the most popular programming languages — according to the TIOBE index [144] — which finds its application in enterprise and web applications which require encryption.

We specifically *do not* select analyses based on the patterns we inject to create the ground truth (see Subsection 8.5.3). This approach allows us to simulate a probable scenario in which the coupling expert does not have detailed knowledge about all patterns the analyses can detect and the patterns that exist in the implementation.

Using these attributes, we select source code analyses from the OWASP list for static source code analyses [241], a community project for software security. This search resulted in the analyses *Find Security Bugs* [18], Semgrep [301] and Snyk [310]. While *Find Security Bugs* and Semgrep are freely accessible open-source analyzers, Snyk is provided as a free version of a commercial tool.

8.5.2. System Selection: Pattern-Search-Based Coupling Approach

The systems for evaluating the accuracy of the pattern-search-based coupling approach are affected by similar requirements as those for the specification-based coupling approach (see Subsection 8.4.2). We require systems for which all of the following is present: architectural models, the implementation in form of source code, specification of security characteristics requiring data encryption for architectural elements representing communication, locations in the implementation which violating the specification of data encryption

and that affect the architectural security. Additionally, we extract the nodes and their communication in the implementation for creating the coupling model with Spoon [249]. Spoon is a tool for analyzing and transforming Java source code which operates on implementations with call-return semantics. Thus, we require the system implementations to use call-return semantics. We did find even less publicly available systems that meet these requirements than for the specification-based coupling approach.

To reduce the complexity of the evaluation, we select the systems from our specification-based coupling approach (see Subsection 8.4.2), as they already meet the requirements of having architectural models and source code available. Furthermore, as we also use the architectural Access Analysis [172], we can reuse the input model of the architectural models already created for the specification-based coupling approach.

Consequently, we select *JPMail*, *CoCoMe* and *TravelPlanner*. Please refer to Subsection 8.4.2 and our replication package [270] for the description of these systems.

JPMail contains system descriptions which can be interpreted to imply *encryption* requirements. Specifically, *JPMail* contains the declassification of the *header* and *body* of the email, which we interpret as the requirement that the data should be encrypted to protect sensitive data during transmission. However, for *TravelPlanner* and *CoCoME* there are no explicit descriptions of *encryption* requirements or implications of such. Thus, for applying *TravelPlanner* and *CoCoME* we derive reasonable *encryption* requirements based on the scenarios described by Katkalov et al. [159] and Greiner and Herda [104]. In the following we describe for each system (1) *how* we specify *encryption* in the architectural model for the coupling and (2) *why* this encryption is reasonable either due to existing descriptions or due to the scenario of the system.

JPMail Tuma et al. [338] use JPMail to evaluate their data flow analysis. In this analysis, they specify *encryption* for the data communication. Seifermann et al. [300] use JPMail to evaluate the DFA, for which they provide a component PCM model. Thus, JPMail is also relevant for our pattern-search-based coupling approach. To represent the *encryption* specification in the architectural model, we annotate every communication link that transmits the data that should be encrypted. According to Tuma et al. [338], the components *SMTP* and *POP3* are in an *Attack Zone* where an adversary can access communication links. In addition, Seifermann et al. [300] provide the components *MailSending* and *MailReceiving* that communicate with *SMTP* and *POP3*. As described by Tuma et al. [338], the data transmitted are expected to be encrypted when sending or receiving emails. Therefore, we annotate every communication link which are connected to resources the components *SMTP* and *POP3* are deployed on with *encryption*. This specification is equal to the specification that the data sent is encrypted.

CoCoMe Altough Greiner and Herda [104] extend CoCoME to *CoCoME with Security* to evaluate information flow analyses, they do not specify *encryption* requirements. However, in the scenario of CoCoME, components communicate sensitive data which *should* be encrypted in a real-world scenario. In particular, the component *CashDesk* sends the credit

EC4	**Description:**	*Whether all data communicated or a subset is affected by vulnerability*
	Categories:	*all data, subset of data*
EC5	**Description:**	*Whether vulnerabilities should propagate through the system or not*
	Categories:	*propagating, non-propagating*

Table 8.5.: Equivalence classes EC4 and EC5 and their categories used for vulnerability injection to create a ground truth for the pattern-search-based coupling approach

card number *cardNumber*, the *storeAccount* and the *amount* of money a customer pays in the selling process to the component *Bank* via the service *requestTransactionService* to handle the payment. The data communicated in this service should be encrypted to protect sensitive data (e.g, the credit card number *cardNumber* and the *storeAccount*). Although every other data communicated is located in the store system, the data communicated to the *Bank* is transmitted over a network that connects external systems. Consequently, we annotate the communication link between the resources the components *CashDesk* and *Bank* are deployed on with *encryption*.

TravelPlanner There is no description of *encryption* requirements for the TravelPlanner. However, the component *TravelPlanner* sends sensitive data to the component *Airline* to book a flight. In particular, the *TravelPlanner* sends the credit card details *ccd* and the *offerId* of the selected flight offer to the *Airline* component to book a flight. For the credit card details *ccd*, the *TravelPlanner* specification [159, 211] specifies that the credit card data has to be declassified before being transmitted to the *Airline* component. We interpret this declassification as the requirement of the credit card data to be encrypted to protect sensitive data during transmission. We also consider it reasonable that the *offerId* is transmitted encrypted to avoid that an adversary can connect the offer to the credit card data. For this purpose, we annotate the communication link between the resources the components *TravelPlanner* and *Airline* are deployed on with *encryption*.

8.5.3. Ground Truth: Pattern-Search-Based Coupling Approach

We now describe the creation of our ground truth by systematically injecting vulnerabilities into the implementations which violate the encryption of data specified in the architectural models. These vulnerabilities are designed based on equivalence classes to guide the injection process. Although we do not use the equivalence classes of the specification-based coupling approach (see Subsection 8.4.3), we continue the sequential enumeration to identify the equivalence classes unambiguously.

8.5.3.1. Equivalence Classes for Injecting Encryption Vulnerabilities

The creation of the ground truth is structured based on three equivalence classes (**EC1**, **EC4**, **EC5**). We reuse **EC1** (introduced in Subsection 8.2.1) to capture whether an injected vulnerability in the implementation has an effect on the architectural analysis result or not. The purpose of **EC4** and **EC5** (summarized in Table 8.5) is to investigate challenges in the pattern-search-based coupling approach arising from the granularity of the annotations and from not considering the semantics of implementation elements.

Equivalence Class EC4 The pattern-search-based coupling approach can only maintain or remove the specification of *encryption* provided to the architectural elements. This is because the results of the source code analyses do only provide information about the existence of vulnerabilities (see Section 5.2). However, multiple data items can be sent in a communication which is implied by the architectural element annotated with the security characteristic *encryption*. When annotating an element with *encryption* mapping to different data, the challenge arises that not all data sent may be affected by a vulnerability invalidating *encryption*. If not all data communicated is affected by a vulnerability, the coupling results in over-approximation because it removes *encryption* for all data communicated.

To capture this challenge, we define equivalence class **EC4**, which describes whether all data communicated is affected by a vulnerability or only a subset of the data is affected. This equivalence class contains the categories *all data* and *subset of data*. A vulnerability belongs to the category *all data* if all data communicated is affected by a vulnerability invalidating *encryption*. A vulnerability belongs to the category *subset of data* if only a subset of the data communicated via a communication link is affected by a vulnerability invalidating *encryption*. We include this equivalence class to illustrate the challenges arising from only being able to maintain or remove *encryption* and the annotation to architectural elements capturing several communications and data.

Equivalence Class EC5 The pattern-search-based coupling approach includes the stop criterion for search strategies which define when the propagation through the coupling metamodel is terminated. This stop criterion is necessary because the pattern-search-based coupling approach is not aware of the semantics of implementation elements (see Section 5.2). In particular, the stop criterion can influence the accuracy of the coupling (see Section 5.3).

To capture the challenges arising from this stop criterion, we define **EC5**, which describes whether an injected vulnerability propagates to one or more security characteristics specified in the architectural model. This equivalence class contains the categories *propagating* and *non-propagating*. A vulnerability belongs to the category *propagating* if the vulnerability must propagate through the system to more than one security characteristic specified in the architectural model. A vulnerability belongs to the category *non-propagating* if the

System	Type of Implementation Vulnerability	# Vul.
TP	Hard-coded Credentials	1
JP	Risky Cryptographic Algorithm	4
CCM	Cleartext Transmission	3
Total		8

Table 8.6.: The Systems, the type of the implementation vulnerability injected and the number of expected architectural vulnerabilities (# Vul.) that must arise after the coupling due to the injected encryption-related vulnerabilities. Systems are denoted with JP (JPMail), TP (TravelPlanner), and CCM (CoCoME).

vulnerability must not propagate through the system to more than one security characteristic specified in the architectural model. We include this equivalence class to illustrate the challenges arising from the approximation performed by the stop criterion in our evaluation.

8.5.3.2. Injection of Encryption Vulnerabilities

Having defined the equivalence classes (**EC1**, **EC4**, and **EC5**), we now describe the specific encryption vulnerabilities we inject into each system in our study design. For this injection we modified the implementations of the systems in our evaluation scenarios to (1) realize the scenarios described in Subsection 8.5.2 and (2) ensure that the vulnerabilities exist in the implementation.

The vulnerabilities we inject are based on three major classes of encryption-related weaknesses detectable in source code.

- **Hard-coded Credentials (CWE-789).** Cryptographic keys or access credentials are hard-coded in source code instead of using secure key management. This weakness manifgtest, for example, if a fixed encryption key is embedded directly in the source code; exposing it to anyone with access to the source code.
- **Risky Cryptographic Algorithm (CWE-327).** A weak or broken encryption algorithm or hashing function (e.g., DES, MD5) is used instead of a strong modern algorithms (e.g., AES-256, SHA-256). Such algorithm can be broken by an attacker with moderate resources, compromising the confidentiality of the data.
- **Cleartext Transmission (CWE-319).** Sensitive data is transmitted without encryption or using an insecure protocol (e.g., HTTP instead of HTTPS). This weakness allows network adversaries to intercept and read the data.

These classes are described by the CWE, the OWASP list of cryptographic failures [239], and by related work in analyses detecting incorrect application of cryptography (e.g., [160, 265, 366]). Furthermore, these classes address different elements in the implementation that are involved in the pattern and the relations to the data affected. Risky cryptographic algorithms are often detected in the *method calls* invoking the cryptographic operations which then are used in encrypting data flowing through the system. Hard-coded credentials

```
public void sendMail (EMailHeader header, EMailBody body){
    // other code ...

        keyGen = KeyGenerator.getInstance("DES");
        keyGen.init(58);
        Cipher cipher = Cipher.getInstance("DES");
        cipher.init(Cipher.ENCRYPT_MODE, keyGen.generateKey());
        byte[] bodyEncBytes = cipher.doFinal(body.toString().getBytes());
        bodyEnc = Base64.getEncoder().encodeToString(bodyEncBytes);

        byte[] headerEncBytes = cipher.doFinal(header.toString().getBytes());
        headerEnc = Base64.getEncoder().encodeToString(headerEncBytes);

    // other code ...

    emailDispatch.dispatch(headerEnc, bodyEnc);
}
```

Listing 8.2: Example of an injected encryption-related vulnerability illustrated on a reduced version of the sendMail method of JPMail.

are often detected in *variable declarations* storing the cryptographic keys which are then used in other parts of the implementation. Cleartext transmissions are often detected in *method calls* to configure or invoke network communication. However, these calls do commonly not establish a direct relation to the data communicated. Using these classes allows us to systematically inject vulnerabilities with a wide range of characteristics into the implementations of our evaluation systems.

We exemplify the injection of a vulnerability into *JPMail* in Listing 8.2. Here, we modify the scenario by encrypting the *header* and *body* of an email with the *risky cryptographic algorithm DES* instead of a strong algorithm like *AES*.

In Table 8.6, we list the number of expected architectural vulnerabilities that must arise after the coupling due to the injected implementation vulnerabilities for each system. Based on the injected implementation vulnerabilities, we expect a total of *eight* architectural vulnerabilities to be reported after applying our coupling approach as follows:

- For *TravelPlanner*, we expect Access Analysis to detect *one* architectural vulnerability.
- For *JPMail*, we expect Access Analysis to detect *four* architectural vulnerabilities.
- For *CoCoME*, we expect Access Analysis to detect *three* architectural vulnerabilities.

TravelPlanner In TravelPlanner, we inject a *hard-coded credentials* vulnerability. We use a hard-coded cryptographic key to encrypt the declassified credit card data *ccd* before sending it to the airline. In contrast, the *offerId* is sent using a key generator with a strong cryptographic algorithm.

The vulnerability has an *effect* (**EC1**) on the architectural analysis because the communication link between the component *TravelPlanner* and the component *Airline* over which

ccd and *offerId* is sent is annotated with *encrypted*. The vulnerability affects a *subset of data* (**EC4**) because the vulnerability does not affect the separately encrypted *offerId*. The vulnerability is *non-propagating* (**EC5**) because the data is only sent to the *Airline* component which is authorized to process the declassified credit card data and the offerId by booking the flight.

Due to the injected implementation vulnerabilities, we expect Access Analysis to detect *one* architectural vulnerability in TravelPlanner after we apply our coupling approach.

JPMail In JPMail, we inject a *risky cryptographic algorithm* vulnerabilities by encrypting the *header* and *body* of emails with a *risky cryptographic algorithm DES* instead of a strong algorithm like *AES* (illustrated in Listing 8.2). Both vulnerabilities have an *effect* (**EC1**) on the architectural analysis because an adversary can break the risky cryptographic algorithm and access the email content. The vulnerabilities affect *all data* (**EC4**) communicated because the same algorithm is used to encrypt all data sent via the communication links.

The vulnerabilities are *propagating* (**EC5**) because the data is sent over multiple communication links. The *MailSendingFacade* (deployed on the resource *AlicePC*) sends the encrypted *header* and *body* to the component *SMTP* (deployed on the resource *MailServer*). The *SMTP* component forwards the encrypted content to the component *POP3* (also deployed on the resource *MailServer*), which sends the data to the component *MailReceivingFacade* (deployed on the resource *BobPC*). The two communication links between *AlicePC* and *MailServer*, and between *MailServer* and *BobPC* are specified with *encryption*. Consequently, the vulnerability affects both annotations.

Due to the injected implementation vulnerabilities, we expect Access Analysis to detect *four* architectural vulnerabilities after we apply our coupling approach. One vulnerability for *header* and one for *body* for the two communication links each.

CoCoME In CoCoME, we introduce a *cleartext transmission* vulnerability by using the insecure HTTP protocol instead of the HTTPS protocol to transmit payment data to the component *Bank*. This vulnerability results in the credit card number (*cardNumber*), the store account (*storeAccount*) and the amount of money a customer pays (*amount*) not being encrypted.

This vulnerability has an *effect* (**EC1**) on the architectural analysis because the communication link between the *CashDesk* and *Bank* components is specified with *encryption* to prevent an adversary to intercept the payment data transmitted. The vulnerability affects *all data* (**EC4**) because HTTP is used for the entire communication. The vulnerability is *non-propagating* (**EC5**) because the data flow terminates at the *Bank* component without further transmission.

Due to the injected implementation vulnerabilities, we expect the Access Analysis to detect *three* architectural vulnerabilities for CoCoME after we apply our coupling approach. Specifically we expect one vulnerability for each affected data items *cardNumber*, *storeAccount*, and *amount*.

8.5.4. Applied Security Relations Model and Coupling Model

We have to apply the pattern-search-based coupling approach to calculate the relations between the models when performing the coupling. For calculating these relations, we apply the *Security Relations* model and the coupling model (see Section 5.2). Thus, to provide insights into the study design for the pattern-search-based coupling approach, we describe the applied *Security Relations* model and the coupling metamodel in the following.

8.5.4.1. Security Relations Model

We create *one Security Relations model* for all evaluation scenarios. The *Security Characteristic* captured in this model is *encryption*. We create one instance of *Weakness* for each weakness class used for creating the ground truth (see Subsection 8.5.3). Therefore, we create a weakness *weak_encryption* (representing the class *Risky Cryptographic Algorithm (CWE-327)*), *unencrypted_connection* (representing the class *Cleartext Transmission (CWE-319)*), and *hard_coded_key* (representing the class Hard-Coded Credentials (CWE-789)). Every instance of *Weakness* violates the *Security Characteristic encryption*.

8.5.4.2. Coupling Metamodel and Model Creation

In our evaluation, we apply a coupling metamodel which is based on a call graph *without* data flow information. This coupling metamodel illustrates the challenge to require approximations when insufficient information is available.

For our study, we perform a simplified process, where the coupling model is first created from the implementation and then the mappings from the architectural model are performed. We construct the coupling metamodel based on a call graph extracted with the program analysis tool Spoon [249]. Spoon extracts information about the given implementation, among others, the method available and which other methods they call. Therefore, we create the coupling metamodel first by creating a *Node* for all methods in the implementation. For every method we then query Spoon regarding the methods it calls. For each called method, we create an *Edge* between the *Node* representing the calling method and the *Node* representing the called method. This way, we create a coupling model representing the call graph of the implementation. Then, we create mappings from the services in the architectural model and the methods in the implementation to the coupling model. We perform this mapping based on the names of the components and classes as well as their respective provided services and methods. The creation of this coupling model is provided in our replication package [270]. We then inspect for every *Edge* whether both nodes are mapped to the architectural model. In this case, we investigate whether the components providing the services are deployed on different resources. If the components are deployed on different resources, we obtain the communication link and map it to the *Edge*. Finally, we investigate if an annotation is applied to the communication link, which

	Access Analysis	DFA	DFA-Ext	JOANA	CodeQL	CodeQL-Ext
Input	1	1	1	1	0.2	1
Output	-	-	-	0.8	0.4	1

Table 8.7.: Results for the *Coverage* of the reference metamodels for the *specification-based* coupling approach by the input and output metamodels of the analyses in the study design for the specification-based analysis coupling. For output metamodels of the architectural analyses, *Coverage* is marked "-" because the reference metamodel for the analysis output is designed only for source code analyses.

specifies *encryption*. If such an annotation exists, we annotate the *Edge* with the *Security Characteristic encryption* of the *Security Relations* model.

We traverse this coupling model with search strategies to determine whether the patterns found by the source code analyses indicate a violation of *encryption* in the architectural model. We use search strategies with a *Stop Criterion* that terminates the traversal as soon as the *Edge* maps to a communication link in the architectural model. With this stop criterion, we avoid traversing the coupling model beyond the point where we can determine whether data communicated is still affected by the vulnerability indicated by the pattern or not.

8.6. Evaluation Results and Interpretation

This section presents the results of our evaluation of the coupling design (**C1**) and coupling approaches (**C2**). We present and interpret the evaluation results of our metrics *Coverage* of the reference metamodels and ICs for the evaluation of *Coverage* (**G1**) in Subsection 8.6.1. In Subsection 8.6.2, we present and interpret the evaluation results of our metrics calculated for the evaluation of *Accuracy* (**G2**).

8.6.1. Result Presentation and Interpretation: Coverage

Here we report and interpret our results to answer the questions **Q1.1**, **Q1.2**, **Q1.3**, and **Q1.4** for our evaluation of goal **G1**. We report the results separately for the coverage of reference metamodels (**Q1.1**, **Q1.2**) and the coverage of ICs (**Q1.3** and **Q1.4**). However, we interpret all results together, to highlight the interplay of the reference metamodels and ICs.

8.6.1.1. Results for Coverage of the Reference Metamodels

Table 8.7 presents our results for the metric $Coverage^{S}_{Ref}$, as we calculated it for six input and three output metamodels in our study of the specification-based coupling approach. The metric is calculated for the four selected analyses (see Subsection 8.4.1) but also for

	Access Analysis	Snyk	Semgrep	*Find Security Bugs*
Input	1	1	1	1
Output	-	1	1	1

Table 8.8.: Results for the *Coverage* of the reference metamodels for the *pattern-search-based* coupling approach by the input and output metamodels of the analyses in the study design for the pattern-search-based analysis coupling. For output metamodels of the architectural analyses, *Coverage* is marked "-" (see Table 8.9)

CodeQL and DFA to investigate the coverage of the reference metamodels by metamodels that are not usable in our coupling. Similarly, Table 8.8 presents our results for the metric $Coverage^{P}_{Ref}$ for the four input and three output metamodels of the analyses in our study of the pattern-search-based coupling approach (see Subsection 8.5.1). In the following, we provide examples for calculating $Coverage^{S}_{Ref}$ and $Coverage^{P}_{Ref}$, especially considering exceptional cases. For the details about the mappings between metaclasses of the reference metamodels and metaclasses of the metamodels of *all* analyses, please see our replication package [270].

We start with the pattern-search-based coupling approach because it is easier to interpret due to homogeneous results. We then continue with the specification-based coupling approach which had some exceptional cases to be discussed.

Results for Coverage of Reference Metamodels: Pattern-Search-Based Coupling Approach
As shown in Table 8.8, all input and output metamodels of the analyses used in our study of the pattern-search-based coupling approach fully cover the reference metamodels ($Coverage^{P}_{Ref} = 1$). We exemplify the calculation of $Coverage^{P}_{Ref}$ for the architectural analysis Access Analysis [172] and the source code analysis Snyk [310].

The reference metamodel for the input metamodel of architectural analyses contains three elements: *Security Characteristic*, *System Element*, and *Annotation*. We determine for each of these three metaclasses at least one conforming metaclass in the input metamodel of Access Analysis ($hold_{Ref} = 1$ for all three metaclasses). The input metamodel of Access Analysis provides the annotation *Encryption* (*Annotation*) to annotate *Linking Resources* (*System Element*). We interpret boolean annotations like *Encryption* to implicitly annotate *Security Characteristics* due to their existence or non-existence (see Subsection 5.2.2). Thus, we determine $hold_{Ref} = 1$ for *Security Characteristic* as well. This results in $Coverage^{P}_{Ref} = \frac{1+1+1}{3} = 1$ for the input metamodel of Access Analysis. The reference metamodels for the input metamodels of source code analysis contains only the metaclass *System Element*, because pattern-search-based source code analyses only search for patters in the implementation without requiring any specification of security characteristics. Snyk searches for patterns in *Method Calls* of Java API methods (*System Element*) or *Field Accesses* (*System Element*). Thus, we determine $hold_{Ref} = 1$ for the only metaclass of the reference metamodel, resulting in $Coverage^{P}_{Ref} = \frac{1}{1} = 1$ for the input metamodel of Snyk.

```
"results": [
   {
       "ruleId": "java/InsecureCipher",
       "ruleIndex": 0,
       "level": "error",
       "message": {
       "text": "The DES cipher used in javax.crypto.KeyGenerator.getInstance (with
           algorithm string \"DES\") is insecure. Consider using AES.",
       "arguments": [
           "[javax.crypto.KeyGenerator.getInstance (with algorithm string \"DES\")](0)"
       ]
       },
       "locations": [
           {
           "physicalLocation": {
           "artifactLocation": {
               "uri": "src/edu/kit/kastel/sdq/coupling/casestudy/jpmail/components/
                   MailReceivingFacade.java",
               "uriBaseId": "%SRCROOT%"
           },
           "region": {
               "startLine": 56,
               "endLine": 56,
               "startColumn": 13,
               "endColumn": 37
               }
           }
       }
       ...
```

Listing 8.3: Excerpt of a pattern detected by Snyk.

The output metamodel for source code analyses in the pattern-search-based coupling approach contains the three metaclasses *ResultEntry*, *Location* and *Pattern*. Listing 8.3 shows an excerpt of a result reported by Snyk for JPMail. Snyk reports a result (*ResultEntry*) as an entry in the array *results* provided in the JSON syntax. Each result contains the *ruleId* (*Pattern*) indicating the pattern that was detected (e.g., "java/InsecureCipher" in Listing 8.3). Furthermore, each results contains a *location* (*Location*) indicating where the pattern was detected in the source code. In Snyk, *location* references the file the pattern was detected in and the lines of code in this file the patterns spans. Thus, we determine $hold_{Ref} = 1$ for all three metaclasses of the reference metamodel for the output metamodel of Snyk, resulting in $Coverage^{P}_{Ref} = \frac{1+1+1}{3} = 1$. The result of Snyk also shows that the reference metamodels are under-specified. For example, the *Result* also contains a *message* to provide a human-readable description for the pattern found, which is not captured in our reference metamodel.

Results for Coverage of Reference Metamodels: Specification-Based Coupling Approach
Table 8.7 shows that full coverage is not achieved for all input and output metamodels of the

analyses in the study involving the specification-based coupling approach. We exemplify the calculation of $Coverage^{S}_{Ref}$ in the specification-based coupling approach for evaluation scenarios where the reference metamodels are fully covered and for evaluation scenarios where they are not. For example, Access Analysis [172] input model is again fully covered ($Coverage^{S}_{Ref} = 1$), but the CodeQL [100, 214] input model is not ($Coverage^{S}_{Ref} = 0.2$).

Each reference metamodel of the specification-based coupling approach defines five elements. We determine for each of the five metaclasses of every reference metamodel at least one conforming metaclass in every input and output metamodel of Access Analysis, DFA, DFA-Ext, and CodeQL-Ext ($Coverage^{S}_{Ref} = \frac{1+1+1+1+1}{5} = 1$). In the information-flow domain, the input metamodel of Access Analysis provides the *InformationFlow* annotation (*Annotation*) to annotate *DataSets* (*Security Characteristic*) to, amongst others, *Parameters* (*System Element*) of *OperationSignatures* (*System Element*). Access Analysis uses its input model as configuration without providing explicit *Configuration* elements (*implicit Configuration*; see paragraph 5.1.2.1). Access Analysis defines unallowed constellations between elements annotated with security characteristics to determine when a vulnerability is reported through a set of Prolog rules (*integrated Security Policy*; see paragraph 5.1.2.1). For example, the Prolog rules define that an adversary with access to a *Location* must be allowed to know at least one *DataSet* that is assigned to a parameter of a component allocated there. Thus, we calculate $hold_{Ref} = 1$ for all five metaclasses of the reference metamodel for the input metamodel of Access Analysis, resulting in $Coverage^{S}_{Ref} = \frac{1+1+1+1+1}{5} = 1$.

CodeQL annotates Java *Parameters* and *Fields* (*System Elements*) with sources and sinks (*Annotations*) within a query file (*Configuration*). A flow is reported if it exists between a source and a sink (*Security Policy*). Detected flows are represented as single entries (*ResultEntry*) which reference the query file and the source code (*Configuration*) to provide the respective *Source* and *Sink* (*Result Entry Element*) containing the respective *Parameters* or *Fields* (*System Element*). CodeQL does not use security characteristics of data (*Security Characteristic*). Thus, we calculate $hold_{Ref} = 0$ for *Security Characteristic* and every element referencing it (i.e., *Configuration*, *Security Policy*, and *Annotation*). Note that this interpretation of CodeQL not using security characteristics of data differs from our interpretation of boolean annotations in the pattern-search-based coupling approach. In the pattern-search-based coupling approach, we interpreted the existence of boolean annotations as implicit security characteristics. We make this distinction deliberately because binary annotations (e.g., *Encryption*) indicate whether a security characteristic holds or not. This interpretation cannot be performed for sources and sinks in CodeQL without further information because the semantics of the annotations differ depending on whether confidentiality or integrity properties should be investigated (see Subsubsection 5.1.2.1). Since CodeQL still uses *System Elements* in its input metamodel, we calculate $Coverage^{S}_{Ref} = \frac{1}{5} = 0.2$. The metric *Coverage* of the reference metamodel for the output metamodels is $Coverage^{S}_{Ref} = \frac{1+1}{5} = 0.4$ because there is no metaclass that conforms to *Security Characteristic* itself, the *Result Entry Element* referencing it, or the *ResultEntry*.

We want to highlight the coverage results for the output metamodel of *JOANA*. The metamodel of *JOANA* does not fully cover the reference metamodel for output metamodels.

```
    entry_point_method: void edu.kit.kastel.sdq.coupling.casestudy.jpmail.Main.main(java.
        lang.String[])
tag: 0
found_flows: true
only_direct_flow: false
flow:
  -
    type: illegal
    attacker_level: low
    source:
      kind: edu.kit.joana.api.sdg.sdgformalparameter
      method:
        class: edu.kit.kastel.sdq.coupling.casestudy.jpmail.components.MailSendingFacade
        name: sendMail
        selector: sendMail(Ledu/kit/kastel/sdq/coupling/casestudy/jpmail/datatypes/
            EMailHeader;Ledu/kit/kastel/sdq/coupling/casestudy/jpmail/datatypes/EMailBody;)V
        return: void
        parameters:
          - edu.kit.kastel.sdq.coupling.casestudy.jpmail.datatypes.EMailHeader
          - edu.kit.kastel.sdq.coupling.casestudy.jpmail.datatypes.EMailBody
      index: 1
      type: java.lang.Object
    source_level: high
    sink:
      kind: edu.kit.joana.api.sdg.sdgformalparameter
      method:
        class: edu.kit.kastel.sdq.coupling.casestudy.jpmail.components.SMTP
        name: dispatchMail
        selector: dispatchMail(Ledu/kit/kastel/sdq/coupling/casestudy/jpmail/datatypes/
            EMailHeader;Ledu/kit/kastel/sdq/coupling/casestudy/jpmail/datatypes/EMailBody;)V
        return: void
        parameters:
          - edu.kit.kastel.sdq.coupling.casestudy.jpmail.datatypes.EMailHeader
          - edu.kit.kastel.sdq.coupling.casestudy.jpmail.datatypes.EMailBody
      index: 1
      type: java.lang.Object
    sink_level: low
            ...
```

Listing 8.4: Excerpt of a report of JOANA.

As shown in the excerpt of a report of *JOANA* for JPMail in Listing 8.4, the *EntryPoint* (identified by the attribute *tag*) contains the *Result Entries* (identified by the attribute *flow*). This structure contradicts the metaclass for the *Result Entries* in our reference metamodel which is described to reference the *Configuration* (i.e., *EntryPoint*). This inverted reference yields $hold_{Ref} = 0$ for *ResultEntry* and, thus, $Coverage_{Ref} = \frac{1+1+1+1}{5} = 0.8$ for the output metamodel of *JOANA*. All other metaclasses of the reference metamodel are fully covered by the output metamodel of *JOANA*, i.e., *Result Entry Element* (*source* and *sink*), *System Element* (identified by the attributes *method* and *index*) and *Security Characteristic* (*source_level* and *sink_level*). However, we apply *JOANA* in our coupling

despite not having full coverage. This initially contradicts our definition of the reference metamodels as interfaces for the coupling. We elaborate on why *JOANA* still is applicable in our coupling when discussing the overall results.

8.6.1.2. Results for Coverage of Integration Conditions

In our evaluation of **Q1.3** and **Q1.4**, we evaluate the coverage of the ICs by the input and output metamodels of the analyses in our specification-based coupling approach. In this evaluation, $hold_{IC}$ evaluated to 1 for all ICs in all sixteen evaluation scenarios on both the model level and the metamodel level (i.e., $Coverage_{IC} = 1$ for all evaluation scenarios). We omit a tabular representation of the results because illustrating this result does not provide further insights. Detailing the results for every IC and every evaluation scenario would not enable further insights, as this comprises 320 applications of $hold_{IC}$ (16 evaluation scenarios for 10 ICs each for the metamodel and model level separately to obtain fine-grained insights). For a detailed description of our calculation of $hold_{IC}$ see the replication package [270].

For illustration, we describe the calculation of $hold_{IC}$ for **IC1** and **IC2** (*correspondence conditions*) and for **IC3** (*constraint condition*) on the evaluation scenario comprising Access Analysis, *JOANA*, and JPMail. On the model level, the *Annotated Source Code* contains the two values *low* and *high* for the metaclass *Level*, both of which have a correspondence to the respective values *low* and *high* of metaclass *DataSet* in the *Annotated Architecture*. Because we require the existence of such a correspondence for **IC1** to hold, we calculate $hold_{IC}(A_{IC}, IC1) = 1$ on the model level. On the metamodel level, the *Annotated Source Code* contains the metaclass *Level* which has a correspondence to the metaclass *DataSet* in the *Annotated Architecture*. Thus, $hold_{IC}(A_{IC}, IC1) = 1$ on the metamodel level as well. Also, all annotated instances of *Parameter* in the *Annotated Source Code* are captured in a correspondence to *Parameters* in the *Annotated Architecture*, for example, between the instance *body* of the metaclass *Parameter* of the PCM and the instance *body* of the metaclass *Parameter* of Java (**IC2**). Similarly, all instances of *EntryPoint* in the *Annotated Source Code* are captured in a correspondence to the implicit configuration of Access Analysis (**IC2**). As a result, $hold_{IC}(A_{IC}, IC2)$ yields 1 on the metamodel and model level because there is at least one correspondence each between (1) instances of the metaclasses *Parameter* of the PCM and a *Parameter* in Java and between (2) the implicit configuration in Access Analysis and instances of *Entrypoint* in JOANA.

Consequently, the metaclasses of the *Annotated Source Code* affected by **IC1** and **IC2** are *Levels*, *Parameters*, and *EntryPoint*. The metaclasses *Source* and *Sink* in *JOANA* specify *Levels* to *Parameters* and are defined in an *EntryPoint* for the analysis of information flows. There are various instances of the metaclasses *Source* and *Sink* (*Annotation*) in the *Annotated Source Code* that reference instances of the metaclasses *Parameter* and *Level* and are referenced by instances of the metaclass *EntryPoint*. In all evaluation scenarios, **IC1** and **IC2** hold. For example, instances of *Source* and *Sink* reference both the instance *high* of the metaclasses *Level* (for which **IC1** holds) and also the instance *body* of the metaclass *Parameter* (for which **IC2** holds). Because **IC3** is specified so that *at least one*

metaclass and *at least one instance* exists, $hold_{IC}(A_{IC}, IC3)$ yields 1 on the metamodel and model level.

8.6.1.3. Result Interpretation

The results of the evaluation show full coverage of the reference metamodels for the pattern-search based coupling approach (i.e., $Coverage^{P}_{Ref} = 1$) by the input and output metamodels of Access Analysis, Snyk, Semgrep and *Find Security Bugs*. All output metamodels of the source code analyses in the evaluation involving the pattern-search-based coupling approach are structured similarly. However, the output metamodels differ in detail, e.g., some analyses provide additional information such as human-readable messages.

The evaluation of the coverage of the reference metamodels described for the specification-based coupling approach show varying results. The reference metamodels are fully covered (i.e., $Coverage^{S}_{Ref} = 1$) by the input and output metamodels of Access Analysis, DFA, DFA-Ext, and CodeQL-Ext.

The coverage metric $Coverage^{S}_{Ref}$ for CodeQL and *JOANA* exhibits values lower than 1. For CodeQL, the metaclass *Security Characteristic* of the reference metamodels caused the metric *Coverage* of the reference metamodels to decrease: $Coverage^{S}_{Ref} = 0.2$ for the input metamodel and $Coverage^{S}_{Ref} = 0.4$ for the output metamodel. The reason for this is that CodeQL uses the specification of sources and sinks *without* security characteristics of data to detect information flows. We cannot interpret the semantics of the source and sink annotations in CodeQL — and, consequently, the security characteristics implied by them — without further information about the security objective (i.e., whether confidentiality or integrity is analyzed). Another aspect to highlight is that CodeQL and the DFA had to be extended to be applied in the coupling (see Subsection 8.4.1). However, the metric $Coverage^{S}_{Ref}$ is 1 for the DFA input metamodel. Our results also show that the input metamodel of *JOANA* is fully covered. However, the reference metamodels are not fully covered by the output metamodel of *JOANA* ($Coverage^{S}_{Ref} = 0.8$), because the reference direction is inverted between the metamodel element *EntryPoint* (*Configuration*) and the *ResultEntry*. Semantically, the reference in the output metamodel of *JOANA* denotes that the counter-example is calculated for the specified *EntryPoint*. Hence, while maintaining the semantics, the reference can be inverted. We demonstrate this by providing an output metamodel fully conforming to our reference metamodel (i.e., $Coverage^{S}_{Ref} = 1$). We applied this metamodel in our coupling instead of the original output metamodel of *JOANA*. In the replication package [270], we provide a transformation that creates instances of this metamodel based on the output of the *JOANA* tooling. That transformation shows the equivalence of both representations while avoiding the modification of the current *JOANA* tooling to provide the new representation.

In summary, when the analyses either annotate information flow-related properties or investigate information flows using security characteristics of data, the metric $Coverage^{S}_{Ref}$ shows that the reference metamodels of the specification-based coupling are indeed fully

covered by the input and output metamodels of the analyses. Therefore, we answer **Q1.1** with *yes*. The reference metamodels of the specification-based coupling approach are covered by the input and output metamodels of architectural and source code analyses to detect security vulnerabilities using information flow-related security characteristics of data. The metric $Coverage^{P}_{Ref}$ also shows that the reference metamodels of the pattern-search-based coupling approach for all analyses used in the evaluation scenarios are fully covered. Therefore, we also answer **Q1.2** with *yes*, the reference metamodels are fully covered by the input and output metamodels of the architectural analyses that specify security characteristics requiring *encryption* and source code analyses that search for patterns indicating vulnerabilities invalidating data encryption. Since all reference metamodels applied in successful couplings are fully covered by the metamodels of the analyses used in our evaluation scenarios of both coupling approaches, we conclude that the reference metamodels are suitable interfaces for the coupling of architectural and source code analyses in the respective coupling approaches.

Furthermore, since we observed $Coverage_{IC} = 1$ on the metamodel level for all evaluation scenarios, we answer **Q1.3** too with *yes*. Finally, since we observed $Coverage_{IC} = 1$ for every successful coupling in our study, we also answer **Q1.4** with *yes*. These metrics show that we can resolve all model elements according to our conditions. Consequently, the ICs are covered on the model level. Since all ICs hold in every successful coupling on the metamodel and model level, we conclude that the ICs are necessary conditions.

We want to highlight two insights gathered in our evaluation of the coverage. First, we extend CodeQL and DFA because they could not be used in the coupling. However, CodeQL showed $Coverage_{Ref} < 1$ because it does not represent security characteristics of data, but DFA showed $Coverage_{Ref} = 1$. Despite the DFA showing full coverage, we had to extend it to be applicable in the coupling. We extended DFA because we could not apply it in the coupling with *JOANA* and CodeQL-Ext. The reason for this is the violation of, amongst others, **IC2** and **IC3**: the security characteristic of the coupling scenario could not be annotated to *System Elements* that are also annotated by *JOANA* and CodeQL-Ext (i.e., parameters). This is the reason why we extended DFA to annotate security characteristics to parameters. This case shows, that conformance to the reference metamodels alone does not guarantee applicability in the coupling because *relations* between analyses may be lacking. Consequently, this case also emphasizes the importance of the ICs for the coupling expert to understand which relations between the analyses must exist.

Secondly, couplings with *JOANA* were successful despite the fact that the output metamodel has no full coverage (i.e., $Coverage^{S}_{Ref} < 1$). To enable the coupling, we provided a transformation to a metamodel that is semantically equivalent to the output of *JOANA* and that conforms to our reference metamodels. This case highlights again the importance of considering the *semantics* captured by the reference metamodels rather than the *syntax* of the metamodels. The case of *JOANA* shows, that a coupling can be established despite $Coverage^{S}_{Ref} < 1$ if transformations for the metamodels can be provided by the *Coupling Expert*, that are *semantically equivalent* to the original metamodels.

Evaluation Scenario	t_p	f_p	f_n	p	r	F_1	F_2
(AccessAnalysis, CodeQL-Ext, CCM)	$\frac{3}{3}$	0	0	1	1	1	1
(AccessAnalysis, CodeQL-Ext, ESS)	$\frac{1}{1}$	0	0	1	1	1	1
(AccessAnalysis, CodeQL-Ext, TP)	$\frac{7}{7}$	0	0	1	1	1	1
(AccessAnalysis, CodeQL-Ext, JP)	$\frac{8}{8}$	0	0	1	1	1	1
(AccessAnalysis, JOANA, CCM)	$\frac{3}{3}$	0	0	1	1	1	1
(AccessAnalysis, JOANA, ESS)	$\frac{1}{1}$	2	0	$\frac{1}{3}$	1	0.50	0.71
(AccessAnalysis, JOANA, TP)	$\frac{7}{7}$	1	0	$\frac{7}{8}$	1	0.93	0.97
(Access Analysis, JOANA, JP)	$\frac{8}{8}$	0	0	1	1	1	1
(DFA-Ext, CodeQL-Ext, CCM)	$\frac{2}{2}$	0	0	1	1	1	1
(DFA-Ext, CodeQL-Ext, ESS)	$\frac{1}{1}$	0	0	1	1	1	1
(DFA-Ext, CodeQL-Ext, TP)	$\frac{4}{4}$	0	0	1	1	1	1
(DFA-Ext, CodeQL-Ext, JP)	$\frac{4}{4}$	0	0	1	1	1	1
(DFA-Ext, JOANA, CCM)	$\frac{2}{2}$	0	0	1	1	1	1
(DFA-Ext, JOANA, ESS)	$\frac{1}{1}$	2	0	$\frac{1}{3}$	1	0.50	0.71
(DFA-Ext, JOANA, TP)	$\frac{4}{4}$	0	0	1	1	1	1
(DFA-Ext, JOANA, JP)	$\frac{4}{4}$	0	0	1	1	1	1

Table 8.9.: *Accuracy* results of study design for specification-based coupling by true positives t_p, false positives f_p, false negatives f_n, precision p, recall r, and the F_β-Scores for $\beta = 1$ and $\beta = 2$ for all evaluation scenarios. Evaluation scenarios in the first column are denoted by (Architectural Analysis, Source Code Analysis, System). For system abbreviations, see Table 8.4.

8.6.2. Result Presentation and Interpretation: Accuracy

Here we report and interpret our results to answer **Q2.1** and **Q2.2** for our evaluation of goal **G2**. We present our results for the accuracy for the specification-based coupling (**Q2.1**) and the pattern-search-based coupling (**Q2.2**) separately due to the different study designs. However, we perform a joint interpretation of the results of both approaches and their respective study designs to come to a comprehensive conclusion.

8.6.2.1. Results for Accuracy: Specification-based Coupling Approach

Table 8.9 presents our results for the metrics precision p, recall r and the F_β-scores for $beta = 1$ and $beta = 2$ (see Subsection 8.2.2) for all sixteen evaluation scenarios obtained by

applying the specification-based coupling approach in our study design. In the following, we discuss one evaluation scenario with precision $p = 1$ and recall $r = 1$ and all scenarios with either $p < 1$ or $r < 1$. Presenting and discussing all results with $p = 1$ and recall $r = 1$ would not provide additional insights. We describe examples for (1) changes in the value of the security characteristic of data due to the coupling, and (2) the resulting vulnerabilities reported by the architectural analyses in the coupling. In our replication package [270], we provide all changes of the security characteristics of data and the resulting architectural vulnerabilities reported by the architectural analysis.

In Table 8.9, the t_p column shows that in all sixteen evaluation scenarios, the architectural analyses reported the vulnerabilities as expected due to the injected information flows (detailed in Subsection 8.2.2). The information flow we introduced in TravelPlanner from the field *ccd* in the class *CreditCardCenter* caused the coupling to modify the values of security characteristic of data that are annotated to the parameter *ccd_decl* in the service *bookFlight* of the component *Airline* from *Airline* to *User*. DFA-Ext and Access Analysis reported the expected vulnerabilities because now data meant only for the *User* is accessible by the *Airline*. The architectural analyses did not report vulnerabilities because of the injected information flow of the class *no effect* as required (detailed in Subsection 8.2.2), even though the value of the security characteristic of data specified for the parameter *details* of the service *addToFlight* of the class *FlightStorage* had be changed from *Airline* to *User*. As seen in the f_n column, our architectural analyses did not report (or miss for the *no effect* category of **EC1**) any vulnerabilities other than the ones expected due to the injected information flows in all scenarios. Consequently, recall is $r = 1$ for all evaluation scenarios.

The f_p column shows that five false positives have been reported after the coupling in three of the sixteen evaluation scenarios. These false positives result in a precision $p < 1$; the remaining scenarios have a precision $p = 1$. In the coupling of Access Analysis with *JOANA* for TravelPlanner, we found one f_p vulnerability ($f_p = 1$), resulting in precision $p = \frac{7}{8} = 0.87$. The vulnerability was reported for the parameter *flightId* in the required *addToFlight* service in the *Airline* component. We investigated this vulnerability further and found that *JOANA* reported information flows from two sources. The first source is the credit card details *ccd* field from the *CreditCardCenter* class. The second source is the *passengerDetails* field from the *TravelPlanner* class. Both fields are specified to contain data with the value of the security characteristic *User*. These flows have also been reported for the parameter *details* in the *addToFlight* service, which resulted in a vulnerability in Access Analysis as expected. The DFA-Ext did not report the vulnerability because it does not consider the respective required interfaces. We discuss the implications of this finding in our interpretation of the results. These false positives result in an F_1-score of 0.93 and an F_2-score of 0.97 in the evaluation scenario including Access Analysis, *JOANA* and TravelPlanner.

For EclipseSecureStorage, the same two f_p vulnerabilities were reported in the couplings of *JOANA* with both Access Analysis and DFA-Ext (i.e., $f_p = 2$ in both evaluation scenarios). These f_p result in precision $p = \frac{1}{3} = 0.33$. One f_p vulnerability was found for the parameter *key* in the provided *internalPut* service of the *SecurePreferences* component; and the other

Evaluation Scenario	t_p	f_p	f_n	p	r	F_1	F_2
(Access Analysis,FindSecBugs, CCM)	$\frac{3}{3}$	0	0	1	1	1	1
(Access Analysis,FindSecBugs, TP)	$\frac{1}{1}$	1	0	$\frac{1}{2}$	1	0.67	0.83
(Access Analysis,FindSecBugs, JP)	$\frac{2}{4}$	0	2	1	$\frac{2}{4}$	0.67	0.56
(AccessAnalysis, Snyk, CCM)	$\frac{0}{3}$	0	3	–	$\frac{0}{3}$	–	–
(AccessAnalysis, Snyk, TP)	$\frac{1}{1}$	1	0	$\frac{1}{2}$	1	0.67	0.83
(AccessAnalysis, Snyk, JP)	$\frac{2}{4}$	0	2	1	$\frac{2}{4}$	0.67	0.56
(Access Analysis, Semgrep, CCM)	$\frac{3}{3}$	0	0	1	1	1	1
(Access Analysis, Semgrep, TP)	$\frac{1}{1}$	1	0	$\frac{1}{2}$	1	0.67	0.83
(Access Analysis, Semgrep, JP)	$\frac{2}{4}$	0	2	1	$\frac{2}{4}$	0.67	0.56

Table 8.10.: *Accuracy* results of study design for pattern-search-based coupling by true positives t_p, false positives f_p, false negatives f_n, precision p, recall r, and the F_β-Scores for $\beta = 1$ and $\beta = 2$ for all evaluation scenarios. The value – signals the result of division by zero. Fractions are omitted for precision and recall if resulting in 1 to improve readability. F_β scores are always provided as decimal values to improve readability. All values are rounded to two decimal places. Evaluation scenarios in the first column are denoted by (Architectural Analysis, Source Code Analysis, System). For system abbreviations, see Table 8.4.

f_p vulnerability was found for the parameter *pathName* in the provided *node* service of the *SecurePreferences* component. Both vulnerabilities arise due to information flows reported by *JOANA* from the parameter *challengeResponse* of the method *SetupRecovery* in the *PasswordManagement* class. We discuss these findings in our interpretation of the results. The precision of $p = \frac{1}{3} = 0.33$ results in an F_1-score of 0.50 and an F_2-score of 0.71.

8.6.2.2. Results for Accuracy: Pattern-search-based Coupling Approach

Table 8.10 shows the true positives, false positives, false negatives, precision, recall and the F_β score with $\beta = 1$ and $\beta = 2$ for each evaluation scenario in the study design for the pattern-search-based coupling.

The application of the pattern-search-based coupling approach to TravelPlanner removed the annotation *Encryption* from the communication link. This is because the source code analyses detected the injected vulnerability of using a hard-coded secret for encrypting the credit card details *ccd* in the *TravelPlanner* class. Based on this detected vulnerability, the coupling approach removed the annotation *Encryption* from the communication link between the resources on which the components *TravelPlanner* and *Airline* are deployed on. This resulted in one true positive $t_p = 1$ vulnerability for the credit card details *ccd*. However, we also obtained a false positive $f_p = 1$ vulnerability for *offerId*. This false positive arised because the encryption of *offerId* uses a generated secret (i.e., not affected

by a vulnerability). However, *offerId* is communicated in the same service as *ccd* in the *TravelPlanner* class. Consequently, *offerId* is communicated through the same communication link which is affected by the removal of the specified *Encryption* security characteristic due to the vulnerability of *ccd*. These true positive and false positive vulnerabilities result in a precision of $p = \frac{1}{2} = 0.50$ and a recall of $r = \frac{1}{1} = 1$. In total, this results in an F_1-score of $\frac{(1+1^2)*\frac{1}{2}*1}{(1^2*\frac{1}{2})+1} = 0.67$ and an F_2-score of $\frac{(1+2^2)*\frac{1}{2}*1}{(2^2*\frac{1}{2})+1} = 0.83$.

For JPMail, every coupling reported two true positive vulnerabilities $t_p = 2$ for the first communication link in the *AttackZone*. These vulnerabilities were reported because the source code analyses detected the injected usage of the risky cryptographic algorithm *DES* to encrypt the parameters *header* and *body* in the class *MailSendingFacade*. With these vulnerabilities, the coupling approach removed the annotation *Encryption* from the communication link between the resources on which the *MailSendingFacade* and *SMTP* components are deployed on. However, the implementation of *SMTP* forwards the data affected by the *risky cryptographic algorithm*, resulting in a violation of the annotation *Encryption* provided to the communication link between the resources on which *POP3* and *MailReceivingFacade* are deployed on. However, the coupling approach did not remove the respective *Encryption* annotation; resulting in two false negatives $f_n = 2$. These false negatives arised because the applied search strategy traversed the coupling metamodel until the first edge is found that maps to the *Annotated Architecture* (see Subsection 8.5.4). These true positive and false negative vulnerabilities result in a precision of $p = \frac{2}{2} = 1$ and a recall of $r = \frac{2}{4} = 0.50$. In total, this results in an F_1-score of $\frac{(1+1^2)*1*\frac{2}{4}}{(1^2*1)+\frac{2}{4}} = 0.67$ and an F_2-score of $\frac{(1+2^2)*1*\frac{2}{4}}{(2^2*1)+\frac{2}{4}} = 0.56$.

For CoCoME, the architectural analysis reported vulnerabilities for *amount*, *creditcarddetails* and *account* sent over the communication link after the coupling with *Find Security Bugs* and Semgrep. We count this outcome as three true positives $t_p = 3$. These vulnerabilities were reported because both source code analyses detected the injected vulnerability of using HTTP instead of HTTPS for communication in the *CashDesk* class. Based on this detected vulnerability, the coupling approach removed the annotation *Encryption* from the communication link between the resources on which the components *CashDesk* and *Bank* are deployed on. These true positive vulnerabilities result in a precision of $p = \frac{3}{3} = 1$, a recall of $r = \frac{3}{3} = 1$, an $F_1 = 1$ and $F_2 = 1$. Snyk does not detect vulnerabilities arising from using HTTP. Consequently, the coupling approach did not perform the removal of the annotation and no architectural vulnerabilities were reported. We count the not-reported vulnerabilities as three false negatives $f_n = 3$, resulting in a recall of $r = \frac{0}{3} = 0$. We cannot calculate precision p, and the F-scores F_1 and F_2 because they result in a division by zero (e.g., $\frac{0}{0+0}$).

8.6.2.3. Result Interpretation

Table 8.9 and Table 8.10 show mixed results regarding the accuracy of the coupled analyses depending on the applied coupling approach.

For the specification-based coupling, three of our evaluation scenarios — all involving JOANA — reported false positive results; thus yielding $p < 1$. All other thirteen evaluation scenarios yielded $p = 1$. We can exclude reasons for these false positives due to unanticipated *data flows*, because CodeQL-Ext did not report the flows leading to the false positives.

We see two possible explanations for these false positives. Either they were caused by implicit information flows we did not comprehend when performing the injection or that they were caused by over-approximative behavior of *JOANA*. Upon further inspection, we found that *JOANA* reported illegal flows for the parameter *key* in the method *internalPut* in EclipseSecureStorage and for the parameter *flightId* in the method *addToFlight* in TravelPlanner. Here, *JOANA* shows over-approximative behavior because, in both evaluation scenarios, information flows for all parameters of the method were reported instead of only one. However, the responsible false positives originated from two different sources: the field *ccd* in the *CreditCardCenter* class and the field *passengerDetails* in the *TravelPlanner* class. We injected a information flow *passengerDetails* to *details* in the method *addToFlight* (see Subsection 8.4.3). Thus, the flow from the field *passengerDetails* can be assumed to be the result of over-approximative behavior of *JOANA*. However, we do not comprehend why *JOANA* reported the information flow from the field *ccd*. Either, there is an implicit information flow we did not consider when performing the injection, or it is again over-approximative behavior of *JOANA*. Additionally, we also found a flow from the field *ccd* to the parameter *details* which we did not knowingly inject. This flow does not result in a false positive because it leaks the same value of the security characteristic *User* as the injected flow from *passengerDetails*. We were unable to replicate the information flow reported by *JOANA* for the parameter *pathName* in the method *node* of EclipseSecureStorage.

Unfortunately, we cannot double-check whether the flows arise from over-approximative behavior or have been injected unknowingly by us because we did not find other source code analyses with working tooling evaluating implicit information flows. In the end, we cannot exclude that our assumption about the accuracy of *JOANA* does not hold (see Subsection 8.2.2). From this we conclude that the output of the source code analysis directly influences the accuracy of the architectural analysis in the coupling. The false positives illustrate a probable scenario in which the *coupling expert* is not fully aware of all the capabilities and details of the applied source code analysis. Still, we also cannot exclude that the information flows reported by *JOANA* arise from implicit information flows in the implementation we unknowingly injected. This highlights the importance of applying source code analyses that can uncover all unallowed information flows in the implementation.

Yet, in all other evaluation scenarios of our evaluation involving the specification-based coupling approach, the coupled analysis found all expected vulnerabilities or did not report those of the class *no effect*. Nonetheless, it was possible to identify vulnerabilities originating from *fields* (category *hidden* (**EC3**) described in Subsection 8.4.3) only by manually providing the specifications. This highlights the importance of the correctness of the specifications provided in the *Completion* step. If developers provide wrong or

incomplete specifications, then in the coupling, the vulnerabilities may not be found or instead, false positive vulnerabilities may be reported, thereby causing precision or recall to decline.

In the investigated evaluation scenarios applying the pattern-search-based coupling approach, the true positives t_p indicate that the coupling *enables* the detection of architectural vulnerabilities originating from vulnerabilities in the implementation. However, the false positives f_p and the false negatives f_n also show that the coupling approach can *negatively affect* the accuracy of the coupled analyses.

The decrease in precision $p < 1$ (i.e., the false positives) in the TravelPlanner shows the negative influence of removing security characteristics specified for architectural elements that define communication of several services and data, such as communication links. Here, *offerId* is reported in an architectural vulnerability ($f_p = 1$) despite not being affected by a vulnerability. This vulnerability was reported because the coupling removes the *Encryption* annotation from the communication link due to the vulnerability in encrypting the credit card details *ccd* with a *hard-coded secret*. This removal resulted in interpreting all data communicated as not encrypted rather than only the data affected by the weakness. From this, we can conclude that the pattern-search-based coupling approach works better for security characteristic specified for more fine-granular architectural elements (e.g., calls or connectors). This limitation results in *software engineers* to be reported with vulnerabilities that do not exist in the final system and are not reported when performing the architectural analysis in isolation.

The false negatives in JPMail decrease the recall $r < 1$. These false negatives occured due to the applied search strategy which traverses the coupling model until the first edge is found that maps to the architectural analysis input model. This resulted in the two false negative vulnerabilities $f_n = 2$ because the subsequent communication link was not considered for inspecting whether it is affected by the implementation-level vulnerability. This finding highlights the importance of considering the semantics of implementation elements to ensure that the coupling can traverse all edges necessary without removing annotations that are not affected by the vulnerabilities detected in the implementation. Note that the architectural analysis in isolation would also not detect these vulnerabilities; again showing that the coupling enables the detection of some vulnerabilities that would not be detected otherwise.

In contrast, the false negatives in CoCoME have been reported because Snyk does not detect flag the usage of HTTP instead of HTTPS as vulnerability. Please note that we used the free version of Snyk, which may have limitations in the patterns it can find compared to the commercial version. This finding illustrates a probable scenario in which the *coupling expert* is not fully aware of all the capabilities and details of the applied source code analysis such as the patterns it can detect.

To summarize, there is a high divergence between the results obtained in the study designs of both coupling approaches, indicating that the effectiveness of the analyses varies significantly based on the methods and scenarios employed. The results involving the specification-based coupling approach are promising as we found 5 f_p compared to a total

of 60 t_p in sixteen evaluation scenarios. Based on this evidence, we answer **Q2.1** with *yes*; that is, the coupled analyses are indeed accurate when applying the specification-based coupling approach to detect architectural vulnerabilities arising from non-compliant information flows involving the specification-based coupling. In contrast, we found 15 t_p, 3 f_p and 9 f_n in a total of nine evaluation scenarios in the evaluation of the pattern-search-based coupling. The f_p and f_n arise (1) by not detecting all architectural vulnerabilities originating from weaknesses in the implementation due to our search strategy and (2) by reporting architectural vulnerabilities that do not originate from the implementation. We specifically used *equivalence classes* in creating the ground truth of the pattern-search-based coupling approach which are based on the challenges we already identified and described in Section 5.2. Therefore, the results of the pattern-search-based coupling approach may be different in other scenarios not affected by these challenges. Still, based on this evidence, we answer **Q2.2** with *no*; that is, the coupled analyses are not accurate when applying the pattern-search-based coupling approach to detect architectural vulnerabilities arising from encryption vulnerabilities in the implementation. We come to this conclusion because the number of true positive t_p results and false results (f_n and f_p) are nearly equal (15 and 12 respectively).

It is to highlight that the evaluation with both study design shows the importance of understanding the capabilities and limitations of the applied source code analyses. The false positives in the specification-based coupling arise due to potential over-approximative behavior of *JOANA*. Similarily, the false negatives in CoCoME in the pattern-search-based coupling arise due to Snyk not reporting the usage of HTTP instead of HTTPS.

Comparing the results of both coupling approaches, it becomes clear that the the accuracy highly depends on the coupling approach itself and the applied source code analyses. Consequently, we must state that reaching the goal **G2** depends on the realization of the applied coupling approaches and the applied source code analyses. However, this encourages us to invest more effort in future work to improve the coupling approaches as discussed in Section 11.3.

8.7. Limitations

Our evaluation of the coupling design and the coupling approaches is subject to several limitations. Future work to address these limitations is discussed in Section 11.3. We continue the enumeration of limitations from the previously stated limitations **L1** - **L15**. For the evaluation of the coupling design and coupling approaches we state six additional limitations **L16** - **L21**.

Limitation 16: Limited Number of Analyses We evaluate our contributions **C1** and **C2** only with a limited number of analyses. Although we selected analyses based on a set of attributes to ensure generalizability (see Section 8.4 and Section 8.5), the limited number of analyses may limit the generalizability of our results. This limitation results in threats

to validity of our evaluation, which we discuss in detail in Section 8.8. The limitation arises, amongst others, because we could not find more analyses with *Working Tooling* (see Section 8.3) that meet our requirements for the evaluation. Still, we argue that our approach of selecting analyses and systems systematically addresses this limitation.

Limitation 17: Limited Number of Systems We evaluate our coupling design and coupling approaches only with a limited number of systems. This limitation arises because we require architectural descriptions, implementations, specifications of values of security characteristics of the coupling scenario and ground truth for the vulnerabilities violating the security characteristics. Finding systems that meet all these requirements is challenging. Also, the effort for creating the ground truth is challenging. We illustrate this challenge through the number of models created in the evaluation (88 in study design applying specification-based coupling approach and 40 in study design applying pattern-search-based coupling approach). In the study design applying the specification-based coupling approach (see Section 8.4), we select our systems TravelPlanner, JPMail, CoCoME, and EclipseSecureStorage based on related work regarding information flow. In the study design applying the pattern-search-based coupling approach (see Section 8.5), we selected the same systems and introduced vulnerabilities resulting in encryption-related non-compliance. For every injected vulnerability, we provide a description to reason about its validity. We argue that our selection of the systems provides a solid basis for evaluating our contributions.

Limitation 18: Subset of Semantics of Values of Security Characteristics in Study Design applying Specification-based Coupling Approach In our study design applying the specification-based coupling approach (see Section 8.4), we only consider a subset of possible semantics of values of security characteristics of data. We selected two systems specifying confidentiality levels with the values *high* and *low* and two systems using specifications of roles with a *disjunctive* lattice. However, there are further semantics of values that we did not explore, which may impact the comprehensiveness of our findings like roles with *conjunctive* lattices or arbitrary security lattices. We already investigate the influence of different semantics of values of security characteristics of data, especially regarding *conjunctive* and *disjunctive* lattices, in the master's thesis of Lehmann [185]. However, we did not include them into the study design for this dissertation because the Access Analysis [172] uses an *integrated security policy* that only supports *disjunctive* lattices and ordered lattices with the values *high* and *low*. Still, using two types of semantics allows us to evaluate whether the coupling approach can handle different semantics of values of security characteristics of data.

Limitation 19: Single Coupling Metamodel in Study Design applying Pattern-search-based Coupling Approach In the study design for the pattern-search-based coupling approach (see Section 8.5), we only consider a single coupling model using a call-graph without dataflow information as basis. This coupling model is sufficient to illustrate the feasibility of our pattern-search-based coupling approach. Furthermore, this coupling model allows

us to illustrate challenges that arise when it only comprises limited information, such as omitting data-flow information. The coupling expert must provide search strategies that approximate the missing information, which can lead to inaccurate results. However, we already performed first investigations in the bachelor's thesis of Traub [336] on the influence of using other structures for the coupling model (e.g., using system dependence graphs [347]).

Limitation 20: Limited Number of Vulnerabilities in Study Design Applying Pattern-search-based Coupling Approach In our study design applying the pattern-search-based coupling approach (see Section 8.5), we only consider a limited number of vulnerabilities that we inject into the systems. Furthermore, many of the vulnerabilities are injected into systems based on rationale given about their plausibility. This limitation arises because finding vulnerabilities for existing systems that fit to our requirements for the evaluation is challenging. However, this limits the comprehensiveness of our findings regarding the pattern-search-based coupling approach.

Limitation 21: Limited Number Of Quality Attributes Evaluated In our evaluation of the coupling design and coupling approaches, we only evaluate the quality of our contributions regarding the *Coverage* of the reference metamodel and ICs, as well as the *Accuracy* of the architectural analysis in the coupling in detecting architectural vulnerabilities arising from non-compliance. However, there are further quality attributes that can be relevant in the context of analysis coupling, such as *Scalability* or *Effort* required to create a coupling. While we discuss the *Effort* argumentatively in Subsection 5.1.5 and Subsection 5.2.6, we do not evaluate it empirically. However, evaluating the *Accuracy* of analyses is widely performed in the context of software analysis (e.g., [12, 111, 150, 257, 300, 348]).

8.8. Threats to Validity

The validity of our evaluation of the contributions **C1** and **C2** is subject to different threats. These threats arise from the aspects common to both study designs and from the aspects specific to each study design. Still, we discuss these threats jointly for both study designs because they share the same evaluation goals, questions, and metrics (detailed in Section 7.1). We organize our discussion by the threat categories *Internal Validity*, *External Validity*, *Construct Validity*, and *Reliability* of Runeson and Höst [282].

8.8.1. Internal Validity

Internal Validity ensures that the factors we investigate in the evaluation remain uninfluenced by factors which the researchers are unaware of. The factors we investigated are the coverage of the reference metamodels, the coverage of the ICs, and the accuracy of the architectural analysis in the coupling.

Threats to Internal Validity of Coverage of the Reference Metamodels We evaluated the coverage of the reference metamodels using our metrics $Coverage^{P}_{Ref}$ and $Coverage^{S}_{Ref}$ in both study designs which are calculated equally. The validity of this metric is influenced by two variables (1) our selection of the analyses, and (2) the mappings created between the reference metamodels. To counter any threat posed by our selection of the analyses in the study designs, we generalize the analyses by constructing a set of attributes and select them such that each attribute from the total set is used at least once in every analysis. These attributes are described specific for analyses applied in the study design involving the specification-based attribute (Subsection 8.4.1) and specific for analysis applied in the study design involving the pattern-search-based coupling approach (Subsection 8.5.1). The correctness of the mappings between the reference metamodels can be negatively influenced by an unclear semantics of the metaclasses. Therefore, to counter this threat, we gained a good understanding of the metaclass semantics (especially of *Annotations*, *Security Characteristics*, *Configurations*, and *Security Policy*) in the specification-based coupling approach by studying the available documentation of Access Analysis [172], DFA [300], *JOANA* [98], and CodeQL [100, 214]. While the complexity of the analyses in the pattern-search-based coupling approach is lower, we also studied the documentation of the analyses which differ from the specification-based coupling approach, namely, *Find Security Bugs* [18], Snyk [310] and PMD [260]. Wherever the documentation was unavailable or proved insufficient to the task, we contacted the authors or maintainers of the analyses.

Threats to Internal Validity of Coverage of the Integration Conditions We evaluated the coverage of the ICs using our metric $Coverage_{IC}$. The metric is influenced by two variables: (1) the transformations created between the (meta)models of the analyses, and (2) the realization of $hold_{IC}$. The transformations created yield, amongst other things, the correspondence models we investigated, and the input and output models like the *Annotated Source Code* we used for that investigation in our evaluation. Consequently, an error in a transformation may negatively impact evaluation results. To counter this threat, we precisely define for each of our sixteen evaluation scenarios, the transformations between the metamodels of the analyses. Creating different transformations also reduces the probability that an error occurs in all transformations. $Coverage_{IC}$ is also influenced by our mapping of the ICs to the metaclasses and references in the input and output (meta)models of the analyses. In short, the metric will depend on our realization of $hold_{IC}$, because a different realization for each case would also result in different and therefore non-comparable results. To counter this threat, we formalize the ICs and automate $hold_{IC}$ based on Unit-Test functions. For each function in our evaluation, we exchange only those metaclasses to be evaluated, while *not changing* the logic of our evaluation (approach detailed in the replication package [270]). By applying the same evaluation logic of $hold_{IC}$ for each IC for all sixteen evaluation scenarios, our automation prevents overfitting the evaluation to one evaluation scenario while thereby increasing the comparability of results.

Threats to Internal Validity of Accuracy Our evaluation of accuracy is mainly influenced by experimenter bias in the creation of a ground truth. This threat is also discussed in related work (e.g., by Tuma et al. [337]). Our ground truth comprises the architectural models, the security specifications, the implementation and all injected vulnerabilities which affect the values of security characteristic of data in our architecture. These vulnerabilities involve the information flow vulnerabilities in the study design applying the specification-based coupling approach and the encryption-related vulnerabilities in the study design applying the pattern-search-based coupling approach. In fact, the implementation in the study design applying the specification-based coupling approach will also influence the values of security characteristic of data if it comprises information flows we injected without intending to. For example, the values of security characteristic of data will be affected by our implementation if we reuse an existing implementation or reimplement the system by using existing information flows. An example is the information flow found by *JOANA* in TravelPlanner system which lead to false positive reports in our evaluation of accuracy. For injecting information flows, we extended the implementation which itself is based on the description of Katkalov et al. [159] for the TravelPlanner system. In these false positive reports cannot exclude that they arise from our extensions but also cannot exclude that they arise from inaccuracies in *JOANA*. To counter this threat, we use (as far as possible) existing models of systems from other authors: for JPMail, the PCM model of Seifermann et al. [300]; for TravelPlanner, the model of Kramer et al. [172] and the behavior description of Katkalov et al. [159]; for CoCoME, the PCM model of Greiner and Herda [104]; and for Eclipse Secure Storage, the implementation of Tuma et al. [337].

This threat can also arise when performing errors in creating the architectural model and the specification. For the study design applying the pattern-search-based coupling approach, this threat arises when performing errors in injecting encryption-related vulnerabilities into the implementation. However, we counteracted this threat two-fold: First, we again used existing models for the system structure and locations of vulnerabilities from related work (e.g., for TravelPlanner [159], for CoCoME [104], and for JPMail [300, 338]) where possible. However, only JPMail included encryption-related details in its existing model. Thus, for the other systems, we used at least the *system structure* from related work. Secondly, we cross-checked the intended injection of encryption-related vulnerabilities by studying current best practices and vulnerabilities described in the *CWE* [327] to ensure that we injected vulnerabilities correctly. We also use different types of encryption vulnerabilities to avoid performing an erroneous injection several times. We did not apply the analyses of our study design to check the correctness of our injection to avoid biasing the evaluation results.

Our evaluation is also threatened by the fact that we are the ones who inject the information flows and encryption vulnerabilities into the implementation to affect the values of security characteristic of data in the architectural input model. Therefore, to counter this threat, in the study design applying the specification-based coupling, we first systematically inject at least one illegal information flow in every system by omitting a declassification. This is common procedure in literature [159, 250, 300, 338], and it has been applied for our case systems TravelPlanner ([159]) and JPMail ([300, 338]). Locations of the declassification has also been described for our case systems EclipseSecureStorage ([250, 337]) and CoCoME

([104]). All remaining information flows are realized similarly. Moreover, to reduce any bias in the injection, the flows are all based on definitions of equivalence classes (given in Subsection 8.4.3). For the pattern-search-based coupling we used vulnerabilities often discussed in literature and described in the *CWE* [327]. We injected the vulnerabilities based on equivalence classes to reduce bias. Additionally, we used locations described in related work where possible. However, we acknowledge that this mitigation only involves JPMail [300, 338] as this was the only system with detailed encryption-related vulnerabilities.

Threats to internal validity can also arise from the quality of the implementations of the coupling approaches themselves. In the specification-based coupling approach, this threat mainly arises from the transformations implemented in the coupling, especially the quality of the retrieval of values of the security characteristic and their integration into the *Annotated Architecture* in the *Parameterization* step. To counter this threat, we use case systems with different semantics of their security characteristic to account for different implementations of transformations. JPMail and EclipseSecureStorage have confidentiality levels with the values *high* and *low* and a *strictly ordered lattice*, while TravelPlanner and CoCoME use roles with the values *User*, *TravelAgency*, and *Airline* with a *disjunctive lattice*. The calculation of the values of security characteristics of data arising from the implementation for confidentiality levels (values *high* and *low*) is less complex than for roles in a *disjunctive lattice*. Thus, using both types of semantics lowers the risk of errors in the implementations of the transformation because (1) the different types of semantics require different implementations, i.e, we cannot use the same algorithms for all transformations, and (2) the lower complexity of confidentiality levels allows us to ensure that at least one implementation is less affected by the threat of low quality transformations. In the pattern-search-based coupling approach, threats arise from the implementation of the search strategies. In particular, errors in selecting wrong paths through the coupling metamodel. We counteract this threat by the applied systems having linear call and data flow structures. Thus, the search strategies are less complex and, therefore, less error-prone. We acknowledge that this mitigation results in a limited generalizability of our findings which we will discuss in Subsection 8.8.2.

8.8.2. External Validity

External Validity concerns the generalizability of our findings beyond just the evaluation scenarios we investigated. In case study research, representativeness may be sacrificed in order to achieve a deeper understanding and exacter realism of the phenomena under study [282]. As we use an evaluation scenario-based design which is similar to case studies, this allows us to gain valuable indicators for scenarios with similar properties. In both study designs, external validity concerns the analyses we selected, the security characteristic we investigated, the architectural models and implementation we used, the values of security characteristics we specified in the evaluation scenarios. These aspects influence the generalizability of our findings in the evaluation of the coverage of

the reference metamodels, the coverage of the ICs, and the accuracy of the architectural analysis in the coupling.

Threats to External Validity of Coverage of the Reference Metamodels First, the external validity of the coverage of the reference metamodel is influenced by the selected analyses because different analyses provide different metamodels for their input and output models. Thus, we may have selected analyses which lead to higher coverage of the reference metamodels than when selecting other analyses. We counteract this threat by selecting analyses based on a set of attributes (detailed in Subsection 8.4.1 for the specification-based coupling approach and in Subsection 8.5.1 for the pattern-search-based coupling approach). We selected analyses such that each attribute from the total set is used at least once in every analysis in the respective study design. In the study design applying the specification-based coupling approach, we selected at least two representatives as architectural analyses and two representatives as static source code analyses. In the study design applying the pattern-search-based coupling approach, we selected three representatives as static source code analyses. One notable aspect is that we selected only one architectural analysis for the pattern-search-based coupling approach as discussed in Section 8.7. We acknowledge that this selection limits the generalizability of our findings to other architectural analyses. However, we argue that the selected architectural analysis (i.e., Access Analysis) is representative for architectural analyses we aim to address. Still, future work should investigate the coverage of the reference metamodels for other architectural analyses to further increase the generalizability of our findings.

Threats to External Validity of Coverage of the Integration Conditions The external validity in our evaluation of the coverage of ICs in the specification-based coupling is influenced by several factors involving the analyses, the selected systems (i.e., the architectural models and implementations) and the specified values of the security characteristics. There may be analyses, architectural models and implementations that lead to our ICs being violated because they do not contain the model elements required by the ICs. Therefore, to counter this threat in the study design applying the specification-based coupling approach, first we classified our analyses based on attributes with regard to our reference metamodels. Classifying the analyses on attributes regarding our reference metamodels is necessary because the ICs are defined based on the reference metamodels. Then we selected (as far as possible) a representative analysis for each combination of attributes. Lastly, we chose systems arguably representative of the application area and reused these with different semantics of their values of security characteristics of data (e.g., roles and confidentiality levels) from related work [159, 300, 338]. To define the ICs we used one isolated case (i.e., Access Analysis, *JOANA*, and the TravelPlanner system) which is also part of the case study and only then we selected the other cases. To avoid overfitting to certain cases, we selected at least two representatives as architectural analyses, two representatives as static source code analyses, and four representatives as systems.

Threats to External Validity of Accuracy Our evaluation of accuracy is also influenced by the selected analyses, systems (i.e., architectural models and implementations), and specified values of the security characteristics. Thus, selecting other analyses, systems, or values of security characteristics may lead to different results. We discuss in Section 8.3, that we assume the source code analysis to be accurate. However, we concluded in Subsection 8.6.2 that this assumption may be false, e.g., in the case of *JOANA* detecting information flows we did not knowingly inject or Snyk not detecting the usage of HTTP instead of HTTPS. Thus, selecting source code analyses with different accuracy may lead to different results than obtained in our evaluation. To counter this threat, we selected two source code analyses in the study design applying the specification-based coupling approach and three source code analyses in the study design applying the pattern-search-based coupling approach. Furthermore, we select a data flow analysis (CodeQL-Ext) and an information flow analysis (*JOANA*) in the study design applying the specification-based coupling approach to facilitate different precision.

One other threat to external validity is the limited generalizability of the vulnerabilities we injected. In the study design applying the specification-based coupling approach, we use at least one security vulnerability that results from information flows for a system in the case study by related work to be representative for the respective domain (for TravelPlanner [159], for CoCoME [104], for JPMail [300, 338], and for Eclipse Secure Storage [76]). Furthermore, we use vulnerabilities described in related work to be representative for the respective domain (for TravelPlanner [159], for CoCoME [104], for JPMail [300, 338], and for Eclipse Secure Storage [250, 337]). In the study design applying the pattern-search-based coupling approach, we use different types of encryption-related vulnerabilities, listed in the *CWE* [327] or in the OWASP list of cryptographic failures [239].

Our findings are also influenced in the study design applying the specification-based coupling approach by the semantics of the values of security characteristics of data we specified in the systems. Investigated by Lehmann [185], different types of semantics may require different transformations in the *Parameterization* step or different execution strategies of the coupling. To counter this threat, we selected systems with different semantics of their values of security characteristics of data (e.g., roles and confidentiality levels) from related work [159, 300, 338]. This selection allows us to account for different implementations of transformations in the *Parameterization* step. However, we acknowledge that we do not cover *all possible semantics* in our study design. One example is the use of *conjunctive lattices* which we do not cover. We discuss this limitation in Section 8.7. However, we argue that our selected semantics are representing common use cases in literature. Confidentiality levels with the values *high* and *low* and a *strictly ordered lattice* are often used in literature (e.g., [63, 224, 338]). Roles with a *disjunctive lattice* are described by Katkalov et al. [159].

Finally, in the study design applying the pattern-search-based coupling approach, the external validity is also influenced by the applied coupling metamodel. We selected a call-graph based coupling metamodel *without* data flows. We performed this selection explicitly to illustrate challenges arising from couplings based on coupling metamodel with limited information. However, we acknowledge that this selection limits the generalizability of

our findings to coupling metamodel comprising also data flows. This limitation is already discussed in Section 8.7 and potential future work to address it in Section 11.3.

8.8.3. Construct Validity

Construct Validity ensures that metrics are capable of answering the evaluation questions. The metrics we applied in the evaluation of **C1** and **C2** are $Coverage_{Ref}$ to evaluate the coverage of the reference metamodels, $Coverage_{IC}$ to determine the coverage of our ICs, and *precision*, *recall* and the F_β-score with $\beta = 1$ and $\beta = 2$ to evaluate the accuracy of the architectural analysis in our coupling. We discuss the appropriateness of each metric to each question in Section 7.1.

Threats to Construct Validity of Coverage of the Reference Metamodels Nevertheless, our evaluation of *Coverage* $Coverage_{Ref}$ is potentially threatened, as well, by the function $hold_{Ref}$. This function may prove inapplicable for determining whether a metaclass of the reference metamodel has a conforming metaclass in the input and output metamodels of the analyses. We are able to exclude this threat because our conformance definition is similar to definitions in similar scenarios (see Subsubsection 4.4.2.3). Consequently, $Coverage_{Ref}$ transfers also to a determination of conformance in similar scenarios. For example, $Coverage_{Ref}$ could also be applied when comparing whether components of a specific architecture cover (i.e., conform to) components of a reference architecture. See Bucaioni et al. [39] for further details.

Threats to Construct Validity of Coverage of Integration Conditions Construct validity also concerns the potential overfitting of the IC. Here, we could have defined ICs only for the specific analyses and systems in our case study, which would have led to an overfitting of the ICs to these analyses and systems. However, we created the ICs based on one isolated case (i.e., Access Analysis, *JOANA*, and the TravelPlanner), which is also part of the case study. For this, we performed an early study in the bachelors' thesis of Häring [116] to get an understanding of the requirements for the coupling of these analyses. Only after defining the ICs, we selected the other cases.

Threats to Construct Validity of Accuracy Our metrics *precision* and *recall* and the F_β-score with $\beta = 1$ and $\beta = 2$ may pose threats by being inappropriate for measuring accuracy. We are able to exclude this threat because *precision* and *recall* are both widely used in related work (e.g. [19, 250, 300, 337, 348]) to measure the accuracy of the analysis under study. The same applies to the F_β-score with $\beta = 1$ (e.g., in [111]). One exemption is the use of the F_β-score with $\beta = 2$, which we introduced to put more weight on *recall* than on *precision*. We argue that this is an appropriate metric for our evaluation because, different studies in security analysis [14, 47] discuss the importance to avoid false negatives (i.e., achieving high *recall*) rather than not reporting false positives. However, we acknowledge that this metric is not widely used in related work.

One other threat to construct validity is the reporting of vulnerabilities which *do not* arise from the coupling. We can exclude this threat because we ensure that the architectural analysis does not report any vulnerabilities (see [250]) when performed in isolation. In this way, we ensure that we consider only those vulnerabilities arising from the coupling.

8.8.4. Reliability

Reliability ensures that results obtained through the data collection and data analysis are independent of particular researchers; in other words, all researchers should also arrive at the same results applying the collection and analysis described in a study. Reliability may be at risk because of the artifacts in the coupling and because of the mappings defined in the evaluation coverage. Although we cannot mitigate this threat, we are able, at least in part, to counter it by providing every evaluation artifact in the replication package [270]. To help ensure the reliability of the calculation of $Coverage_{Ref}$, we provide all metamodels used in the analyses as well as their mappings to the reference metamodels in the replication package [270]. Therefore, researchers can validate whether they arrive at the same results when calculating $Coverage_{Ref}$ based on our provided artifacts. Furthermore, we increase reliability by automating the calculation of $hold_{IC}$, which determines the coverage of the ICs (detailed in the replication package [270]). In this way, we enable other researchers to reproduce our results by applying this automation to other metamodels and models in an established coupling. As one further assurance of reliability of the coverage evaluation, we provide all input models, output models, correspondence models, and their metamodels used in the coupling of the analyses. With this, researchers can further understand our procedure and adapt it to their models and mappings. To help ensure the reliability of the accuracy evaluation, we provide all of the following in the replication package [270]: the architectural analyses, the source code analyses, the security characteristics under investigation, the implementations, and all details about the injected information flows and their expected impact on the architectural analysis. In particular, we argue that our injection of information flows by removal of declassification mechanism is repeatable and therefore reliable because it has been used in related work ([159, 300, 338]). We also describe the calculation of the security characteristics in the specification-based coupling approach in the replication package [270]. With this description, researchers can verify whether the calculation is reasonable and repeat the calculation for our implementations to verify our findings. Our replication package [270] also contains an executable version of the analyses applied in our evaluation in a virtual machine. With this virtual machine, researchers have the possibility to execute the analyses even when the employed versions or dependencies for them are no longer available. Our metrics *Coverage* and *Accuracy* (detailed in Section 7.1) give reasonable evidence, which, in turn, reduces the necessity for interpretation. Thus, due to the study designs in Section 8.4 and Section 8.5, there is hardly an interpretation that may lead a researcher to another conclusion.

8.9. Summary and Outlook

In this chapter, we presented the evaluation of our coupling design (**C1**) and our coupling approaches (**C2**). We evaluated both contributions together because evaluating the coupling design requires instantiating it with specific coupling approaches. The evaluation goals were to evaluate (1) the *Coverage* of the reference metamodels and ICs defined in the coupling approaches (**G1**), and (2) the *Accuracy* of the coupled analyses (**G2**).

For the evaluation, we applied an *evaluation scenario*-based design which is similar to a case study but highlights that we do not use real-world cases. Each evaluation scenario comprises a *coupling scenario* (i.e., an architectural analysis, a source code analysis and a security characteristic to be analyzed for non-compliance) and a *system* to be analyzed. The systems comprise an architectural description, an implementation and a specification of values of the security characteristic to be evaluated in the coupling. Case studies — and in extend evaluation scenarios — are one of the preferred methods to evaluate notations of models [26] and to evaluate the accuracy of security analyses (e.g., [19, 250, 300, 348]).

We first specialize the coverage metrics for the reference metamodels and ICs defined in the coupling approaches to evaluate the coverage of these structures by the analyses applied in the coupling. Then, we outline how we obtain a ground truth for the evaluation of the accuracy of the coupled analyses and the classification of the results obtained by applying the coupling. However, this ground truth must be created for both coupling approaches separately because they require different security characteristics and vulnerabilities to be evaluated, which makes a joint study design infeasible. Furthermore, we describ requirements common for the evaluation of both coupling approaches and the specific study designs for each coupling approach. Based on these study designs, we present and interpret the results of the evaluation regarding the coverage and accuracy of both coupling approaches. Finally, we discuss limitations and threats to validity of our evaluation. We summarize these topics in the following.

Specialization of Coverage Metrics We specialized the coverage metric for the reference metamodels $Coverage_{Ref}$ (**M1.1.1**) by defining a condition $cond \in COND$ for every metaclass in the reference metamodels applied. Each condition states that the metamodels of the architectural analyses or source code analyses — used as input artifacts A for the coverage metric — must have at least one metaclass that conforms to the metaclass of the reference metamodel according to Definition 6. This conformance is determined by the function *hold*. We also specialized the coverage metric for the ICs $Coverage_{TC}$ (**M1.3.1**) by defining conditions $cond \in COND$ for every IC. This condition states that the metamodels and models of the analyses in the coupling must provide metaclasses or their instances which are related according to the IC. Determining these relations is supported by the application of correspondence models established in the coupling process. Thus, the artifacts used for the coverage metrics for the ICs are the input and output (meta)modes of the analyses, and the correspondence models established in the coupling process. We defined when *hold* determines a condition to hold by accounting for the IC to be either a *correspondence condition* or a *constraint condition*.

Ground Truth and Results Classification To evaluate the accuracy of the coupled analyses, we required a ground truth of architectural vulnerabilities obtained from the coupled analysis which arise from non-compliance in implementations. This ground truth defines the expected results of the coupled analyses and allows us to classify the results obtained by applying the coupling into true positives, false positives and false negatives.

For defining the ground truth (i.e., the expected results of the coupled analyses), we require (1) an architectural system description and specification, (2) an implementation corresponding to this description, (3) known vulnerabilities in the implementation that lead to non-compliance regarding the security characteristic to be evaluated in the coupling, and (4) the architectural vulnerabilities arising from the non-compliance in the implementation. In case of the specification-based coupling approach also needs the specification of values of the security characteristic. To the best of our knowledge, there is no such a ground truth in related work. Therefore, we reuse the architectural models, implementation, specifications and vulnerabilities from related work where possible and create the missing artifacts ourselves. In particular, we state that we create the architectural models in a way that no architectural vulnerability arises when performing the architectural analysis in isolation. This ensures that the architectural vulnerabilities we find in the coupled analyses indeed arise from non-compliance in the implementation and not from the architectural design itself.

Furthermore, we systematically inject vulnerabilities into the implementation of the systems to create the required ground truth for our evaluation. For injecting the vulnerabilities, we use equivalence classes to ensure that the injected vulnerabilities influence the coupling and the results of the coupled analyses in a similar way. We introduce the first euqivalence class which is common to both coupling approaches. This equivalence class captures whether a vulnerability which leads to non-compliance regarding the security characteristic leads to a report of an architectural vulnerability by the coupled analyses or not.

Based on this approach to create the ground truth, we provide a comprehensive classification of obtained architectural vulnerabilities when applying the coupling into true positives, false positives and false negatives. This classification allows to calculate precision, recall and the F_β-score for the coupled analyses in the evaluation of both coupling approaches. In this definition, we have to provide different definitions for true positives, false positives and false negatives because of the first equivalence class which captures whether a vulnerability leads to a report of an architectural vulnerability or not.

Requirements and Assumption We define two requirements and one assumption for the evaluation of both coupling approaches.

The first requirement states that the description of the metamodels and models in the evaluation with the Eclipse Modeling Framework (EMF). We state this requirement because many design-level analyses use the Eclipse Modeling Framework (EMF) for their metamodels and models, and we want to ensure that we can apply the coupling approaches

with these analyses. We also state that we reconstruct the metamodels of the analyses in the evaluation with the EMF if they are not available.

The second requirement states that the analyses applied in the coupling must have *working tooling*. This requirement is necessary as we have to apply the coupling to evaluate the accuracy of the coupled analyses, which requires applying the analyses to obtain results.

For the applied source code analyses, we assume that they are accurate in detecting the vulnerabilities we inject into the implementation. This assumption is necessary to define the ground truth for the evaluation of the accuracy of the coupled analyses. Based on evidence from related work, we are aware that this assumption does not hold in practice. However, we argue that this assumption allows us to define a ground truth without overfitting to the source code analyses, and to account for probable scenarios where coupling experts are not fully aware of all details about the analyses.

Study Design: Specification-based Coupling Approach In the study design applying the specification-based coupling approach, we couple *two* architectural analyses with *two* specification-based source code information flow analyses each. These analyses are selected based on a set of *six* attributes. We could apply the architectural analyses *Access Analysis* [172] and the source code analysis *JOANA* [98] without further intervention. However, due to the lack of analyses with *working tooling*, we had to extend the architectural analysis *Data Flow Analysis* [300] and the source code analysis *CodeQL* [100, 214] to be applicable with our coupling. The DFA did not annotate security characteristics of data to parameters which was necessary for establishing the coupling. However, the DFA had configurable security policies that could be extended to account for security characteristics of data annotated to parameters. Therefore, we were able to extend the DFA to use security characteristics of data specified to parameters while maintaining its original functionality. CodeQL is able to detect data flows using parameters and fields but does not use specifications of security characteristics of data for this detection. However, CodeQL could be extended with custom security policies and elements. Thus, we were able to extend CodeQL to account for security characteristics of data specified to parameters and fields while maintaining its original functionality. We did not modify the implementations of these analyses and extended the (meta)models non-invasively. Consequently, the architectural analyses applied in the coupling were the *Access Analysis* and the extended *DFA* (DFA-Ext). The source code analyses applied in the coupling were *JOANA* and the extended *CodeQL* (CodeQL-Ext).

We use the systems TravelPlanner, JPMail, CoCoME, and EclipseSecureStorage in the evaluation of the specification-based coupling approach. We select these systems based on related work regarding information flow and the availability of architectural models, implementations and specifications of values of security characteristics. Furthermore, these systems allow us to investigate different semantics of values of security characteristics of data, which is relevant for the comprehensiveness of our evaluation.

We investigate non-compliance with *one* security characteristic in the coupling scenario for each system. This resulted in a total of sixteen evaluation scenarios. For creating the

ground truth, we inject vulnerabilities into the implementation of the systems which lead to non-compliance regarding the security characteristic. We injected vulnerabilities in the implementation of the systems which lead to a total of *60* architectural vulnerabilities we expect to be reported by both architectural analyses. We injected these vulnerabilities by selecting at least one vulnerability in the implementation that has been described in related work.

Study Design: Pattern-Search-Based Coupling Approach In the study design applying the pattern-search-based coupling approach, we couple *one* architectural analysis with *three* pattern-search-based source code analyses that were able to detect patterns indicating vulnerabilities in the implementation that invalidate the encryption of data. The architectural analysis applied in the coupling was again the *Access Analysis* [172]. The source code analyses applied in the coupling are *Snyk* [310], *Semgrep* [301] and Find Security Bugs [18]. We select these analyses based on a set of *four* attributes. We investigate non-compliance with *one* security characteristic in the coupling scenario and apply the coupling to *three* systems.

We also use the systems TravelPlanner, JPMail, CoCoME in the evaluation of the pattern-search-based coupling approach. We select these systems due to a lack of fitting systems in related work with the required artifacts for creating the ground truth for the evaluation of the pattern-search-based coupling approach. JPMail contains an architectural design, an implementation and the description of encryption which alled us to inject vulnerabilities leading to non-compliance regarding encryption. TravelPlanner and CoCoME contain an architectural design and an implementation but no description of encryption. However, we have seen potential for injecting vulnerabilities leading to non-compliance regarding encryption in these systems due to the communication of sensitive data (although not explicitly described in the specifications). Additionaly,a goal of the study design is illustrating challenges stemming from the techniques applied by the pattern-search-based analysis coupling. These systems allow us to illustrate challenges arising from the pattern-search-based coupling approach, such as annotating architectural elements that represent communication of multiple data, or propagating the influence of vulnerabilities in the implementation through the system. The selection of the analyses and the systems results in a total of *nine* evaluation scenarios.

We inject vulnerabilities in the implementation of the systems which lead to a total of *eight* architectural vulnerabilities distributed over all systems. The vulnerabilities had been injected by selecting major encryption-related vulnerability types in the implementation that are described by the CWE, the OWASP list of errors when applying cryptography [239], and analyzed by analyses in related work.

Finally, we present the *Security Relations* model and the coupling metamodel we applied in the coupling for the pattern-search-based coupling approach.

Results and Interpretation: Coverage We present the results of the evaluation regarding the coverage of the reference metamodels and ICs (**G1**) for both coupling approaches

jointly. This joint presentation allows to compare the results of both coupling approaches and to interpret them in relation to each other.

The results of the evaluation showed that the reference metamodels of the specification-based coupling approach are covered by Access Analysis [172], DFA [300], CodeQL-Ext, and DFA-Ext. CodeQL [100, 214] and *JOANA* [98] do not fully cover the reference metamodels of the specification-based coupling approach. CodeQL does not annotate security characteristics and the annotations also cannot be interpreted for a specific semantic. Thus, CodeQL does not cover the reference metamodels related to the specification of values of security characteristics. *JOANA* provides an references between the *Configuration* and the *Result Entry* in the output metamodel which is inverted regarding the reference defined in the reference metamodel. The reference metamodels of the pattern-search-based coupling approach are covered by all applied analyses; namely Access Analysis [172], *Snyk* [310], *Semgrep* [301] and Find Security Bugs [18]. One particular case is the input metamodel of the Access Analysis, which uses an annotation representing whether data sent over a communication link is encrypted or not with out a dedicated security characteristic. However, the specific semantic and the binary value range of this annotation allows the interpretation of existence of security characteristic because the meaning of the annotation is clear. This contrasts CodeQL in the specification-based coupling approach, where the annotations cannot be interpreted regarding a specific semantic. From these results regarding the coverage of the reference metamodels, we conclude that the reference metamodels of both coupling approaches are indeed feasible interfaces for the coupling.

The ICs defined in the specification-based coupling approach are covered by all applied analyses; namely Access Analysis [172], DFA-Ext, *JOANA* [98] and CodeQL-Ext. Consequently, we conclude that the ICs defined in the specification-based coupling approach are indeed necessary but not sufficient conditions to achieve the coupling with these analyses.

We present two notable insights obtained from the evaluation of the coverage of the reference metamodels and ICs. Firstly, CodeQL and DFA show a notable case in our coupling approach which highlights the importance of the joint consideration of the the reference metamodels and ICs for the coupling. Although both analyses had to be extended to be applicable in the coupling, the DFA fully covers the reference metamodels of the specification-based coupling approach while CodeQL does not. This result illustrates that the coverage of the reference metamodels alone does not suffice for an analysis to be applicable in the coupling, but that the ICs efined in the specification-based coupling approach are indeed necessary to achieve the coupling.

Secondly, although *JOANA* does not fully cover the reference metamodels of the specification-based coupling approach, it is still applicable in the coupling because it covers the ICs defined in the specification-based coupling approach. This result illustrates the importance to consider the semantics of the reference metamodels and the metamodels of the analyses rather only the syntax.

Results and Interpretation: Accuracy The results of the evaluation regarding the accuracy of the coupled analyses (**G2**) for both coupling approaches are presented separately because they are based on different study designs and different ground truths. However, we then interpret the results of both coupling approaches jointly to come to a comprehensive conclusion. able and that the ICs are indeed necessary to achieve the coupling.

For each study design, we present the results and the calculation of precision (**M2.1.1**), recall (**M2.1.2**) and the F_β-score (**M2.1.3**) for $\beta = 1$ and $\beta = 2$. In the study design applying the specification-based coupling approach, we found 60 true positive results and five false positive results in the 16 evaluation scenarios. No false negative results had been reported. The false positive results arose in three evaluation scenarios; all involving the analysis *JOANA*. We found indications that the assumption about its accuracy does not hold. In the affected evaluation scenarios, *JOANA* reported information flows to all parameters in the same methods instead of only to the parameter we targeted in the injection. However, we also found information flows from other parameters and fields we did not comprehend. We could not investigate the correctness of the flows reported by *JOANA* because the false positives only occurred through implicit information flows which CodeQL-Ext is not able to detect. This led us to the result that we cannot dismiss the possibility of unknowingly injecting information flows we did not comprehend.

The evaluation using the study design applying the pattern-search-based coupling approach showed more diverse results. In the nine evaluation scenarios, we found a total of 15 true positive results, three false positive results and nine false negative results. Three false positives occurred in the three evaluation scenarios involving the same system. In this system, only one of two parameters in a method call was designed to be involved in the injected architectural vulnerability. However, the removal of the specified value of the security characteristic by the coupling approach resulted in the report of a vulnerability for both parameters. This case illustrated the challenge arising from the handling of security characteristics specified for architectural elements that represent communication of multiple data. Six false negatives occurred in three evaluation scenarios involving the same system. There, the applied coupling approach did not propagate the encryption weakness through the system which would be necessary due to the implementation. This outcome illustrated the challenge that our pattern-search-based coupling approach does not consider the semantics of implementation elements.

In one evaluation scenario involving the source code analysis *Snyk*, three false negatives had been calculated. In this scenario *Snyk* did not report the injected vulnerability (using HTTP instead of HTTPS) while *Find Security Bugs* and Semgrep reported them. This case illustrates a probable scenario where a coupling expert is not fully aware of the patterns detected by the source code analyses.

In summary, our evaluation showed a diverse result for the accuracy of both coupling approaches. The specification-based coupling approach showed reported all 60 expected architectural vulnerabilities in the evaluation scenarios and only 5 false positives. In contrast, the evaluation of the pattern-search-based coupling approach showed only in two evaluations precision and recall values of 1, while in others these values were significantly lower. This result indicates that the accuracy of the coupling depends on the challenges in

the realizations of the coupling approach. Still, both approaches showed the feasibility of the coupling to detect architectural vulnerabilities arising from non-compliance in the implementation, which is a major result of our evaluation.

Limitations We discuss six limiations of our evaluation of the coupling design and coupling approaches. Two limitations arise from the limited number of analyses and systems we could apply in the evaluation, which may limit the generalizability of our results. Furthermore, we do not consider all possible semantics of values of security characteristics of data in the study design applying the specification-based coupling approach, which may limit the comprehensiveness of our findings. In the study design applying the pattern-search-based coupling approach, we only consider a single coupling model, and also only a limited number of vulnerabilities, which may limit the comprehensiveness of our findings. Finally, we only evaluate a limited number of quality attributes. However, we argue that we discuss accuracy as one major quality attribute in the context of software analysis.

Threats to Validity We discuss threats to validity of our evaluation, structured by the threat categories *Internal Validity*, *External Validity*, *Construct Validity* and *Reliability* of Runeson and Höst [282].

Internal Validity concerns factors that may influence our evaluation results unknowingly. For the coverage of the reference metamodels, internal validity is influenced by our selection of analyses and the mappings between reference metamodels. We counter this threat by generalizing analyses through attributes and ensuring clear semantics of metaclasses through studying documentation and contacting authors. For the coverage of ICs, internal validity is influenced by the transformations between (meta)models and our realization of $hold_{IC}$. We counter this threat by formalizing ICs, automating $hold_{IC}$ through unit tests, and applying consistent evaluation logic across all evaluation scenarios. For accuracy, internal validity is mainly threatened by experimenter bias in creating the ground truth (including architectural models, specifications, implementations, and injected vulnerabilities). We counter this threat by reusing existing models and implementations from related work where possible, systematically injecting vulnerabilities based on equivalence classes, using locations for vulnerability injection when possible from literature, and ensuring the architectural analysis reports no vulnerabilities in isolation. Internal validity is also threatened by the quality of coupling implementations, particularly transformations. We address this by using case systems with different semantics of security characteristics (e.g., confidentiality levels and roles with different lattice structures) and linear implementations for call and data flows in the pattern-search-based coupling approach.

External Validity concerns the generalizability of our findings. For the coverage of reference metamodels, external validity is influenced by selected analyses, as different analyses provide different metamodels. We counter this threat by selecting analyses based on attributes, ensuring each attribute is used at least once, and selecting multiple representatives (two architectural analyses and two source code analyses for specification-based coupling, one architectural analysis and three source code analyses for pattern-search-based coupling).

For evaluating the coverage of ICs, external validity is influenced by analyses, selected systems, and specified values of the security characteristic under investigation. We address this threat by classifying analyses based on attributes regarding reference metamodels, selecting representative analyses for attribute combinations, and using systems representative of application areas with different security characteristic semantics from related work. For evaluating accuracy, external validity is influenced by selected analyses, systems, and security characteristic values. We counter this threat in the study design applying the specification-based coupling approach by using systems and vulnerabilities described in related work as representative for their domains, using systems with different semantics of the security characteristic under investigation (i.e., roles and confidentiality levels) and selecting analyses with different precision (e.g., data flow vs. information flow analyses). We address this threat in the study design applying the pattern-search-based coupling approach by using different types of vulnerability classes described in CWE and OWASP. However, we acknowledge limitations in the generalizability due to limited semantics coverage (e.g., not covering conjunctive lattices) and the use of call-graph based coupling metamodel without data flows in the pattern-search-based approach.

Construct Validity ensures that our metrics are capable of answering the evaluation questions. Our evaluation of the coverage of the reference metamodels may be threatened by the function $hold_{Ref}$ being inapplicable for determining conformance. We exclude this threat because our conformance definition is similar to definitions in comparable scenarios. Our evaluation of the coverage of the ICs may also be threatened by overfitting the ICs to the specific analyses and systems in our case study. We address this threat by defining the ICs based on one isolated case (Access Analysis, *JOANA*, and TravelPlanner) before selecting other cases. Construct validity of the accuracy evaluation may be threatened by the metrics *precision*, *recall*, and F_β-score being inappropriate for measuring accuracy, and by architectural vulnerabilities that do not arise from the coupling. We exclude the threat regarding the selected metrics *precision*, *recall*, and F_β-score are appropriate because they are widely used in related work for measuring analysis accuracy. We counter the threat regarding architectural vulnerabilities that do not arise from the coupling by creating architectural models in a way that no architectural vulnerability arises when performing the architectural analysis in isolation.

Reliability ensures that results are independent of particular researchers and reproducible. This may be at risk due to artifacts in the coupling and mappings defined in the evaluation. We counter this threat by providing all evaluation artifacts in the replication package [270], including all metamodels and their mappings, automating the calculation of $hold_{IC}$, providing all input/output models and correspondence models, and providing executable versions of analyses in a virtual machine. For accuracy evaluation, we provide architectural analyses, source code analyses, security characteristics, implementations, and details about injected vulnerabilities. The injection of vulnerabilities by removal of declassification mechanisms is repeatable as it has been used in related work. Our metrics give reasonable evidence with minimal need for interpretation, reducing the likelihood of different researchers reaching different conclusions.

Outlook In the next chapter (Chapter 9), we present the evaluation of *SecLan* (**C3**). This evaluation is independent of the evaluation of the coupling design and coupling approaches. In this evaluation, we inspect (1) the *correctness* and *completeness* of the *SecLan* conceptual model (**G3**) and (2) the *correctness* of the relationships calculated by *SecLan* (**G4**).

9. Evaluating SecLan

> **Literature:** This chapter is based on the following (co-) authored publications: "*Can I Check What I Designed? Mapping Security Design DSLs to Code Analyzers*" [254] (major revision under review in the journal *ACM Transactions on Software Engineering and Methodology* at the date of submission of the dissertation)

This chapter presents the evaluation of our third contribution **C3** – *SecLan* (see Chapter 6). The evaluation is based on surveys and interviews with experts because validating the *SecLan* conceptual model or the relationships calculated would be affected by bias (see Section 7.2). For this evaluation, Section 9.1 presents the design of the surveys, the design of the expert interviews, and the selection of participants. This study design contrasts the study design for evaluating the coupling design and coupling approaches which comprises the construction of the evaluation scenarios. We then present and discuss the findings of our surveys and interviews in Section 9.2. The limitations of our study design are discussed in Section 9.3. We discuss the threats to validity stemming from these limitations in Section 9.4.

9.1. Study Design for Evaluating *SecLan*

As outlined in Section 7.2, we validate the *correctness and completeness* of the *SecLan* conceptual model (**G3**) by using surveys and qualitatively explore the relationships calculated by *SecLan* (**G4**) by performing semi-structured interviews.

To answer the questions regarding these goals (see Section 7.1), we use the instances of the *SecLan* conceptual model for 66 security DSLs and 33 analyzers containing 408 security checks created during the synthesis of the *SecLan* conceptual model. We then apply *SecLan* to these instances to calculate relationships between the security DSLs and the security checks of the analyzers. All created instances can be viewed in our *SecLan* repository[1] or found in our replication package [270].

To build a common understanding of the rationale behind the survey and interview design, we first discuss the expert selection in Subsection 9.1.1. Then, we detail the survey design in Subsection 9.1.2. Finally, we describe the interview design in Subsection 9.1.3.

[1] `https://gravity-tool.org/seclan-repository/index.html`

Subsection 9.1.4 details how we calculate the metrics Krippendorff's α, the Percentage Agreement PA, and Gwet's $AC1$ for inter-rater reliability (see Section 7.1).

To enable practical applicability of *SecLan* for our evaluation, we created tool support to describe security DSLs and analyzers according to the *SecLan* conceptual model and to calculate the relationships between them automatically. We provide the tool in the version applied for the evaluation in our replication package [270]. This tool is then used to present the created instances of the *SecLan* conceptual model to the participants of our surveys and interviews. Furthermore, we use the tool to calculate and present the relationships between the security DSLs and analyzers used in the qualitative exploration.

9.1.1. Participant Selection

To better understand the questions designed for the surveys and interviews, we start by discussing the selection of participants. Validating the correctness and completeness of the *SecLan* conceptual model and its instances, as well as validating the relationships calculated by *SecLan* requires expert knowledge. We determine the authors of research papers of security DSLs and analyzers as well as developers and maintainers of their tooling as suitable participants for our surveys and interviews. We call these participants collectively *experts* in the remainder of this chapter. Ananieva et al. [15] also follows the approach of asking experts to investigate the correctness and completeness of their conceptual model.

The authors of research publications about security DSLs and analyzers are suitable participants because they usually have deep knowledge about the concepts and relations in their respective domain. We aim to invite all 304 authors of the 66 security DSLs and 33 analyzers we studied to participate in our survey (186 and 118 respectively). However, we only found current contact email addresses for 132 of them, to which we send the invitation. We invite developers and maintainers of the respective security DSLs and analyzers (e.g., looking up the most active developers in GitHub repositories) if we found applicable contact email addresses.

9.1.2. Survey Design for Evaluating *SecLan*

We use surveys to answer the questions about the *correctness and completeness* of the *SecLan* conceptual model (**G3**) to counteract threats arising from creating the *SecLan* conceptual model or its instances. For this purpose, we design two online surveys using the platform *LimeSurvey*[2]. The first survey targets experts in security DSLs and the second targets experts in analyzers.

The surveys are structured similarly by containing an introduction, two core parts and a follow-up request for participating in our interview. The first core part investigates

[2] `https://www.limesurvey.org/`

whether the concepts and relations between these concepts as well as the specific *Element Types* and the relation between these *Element Types* in the *SecLan* conceptual model fit the understanding of the experts. The second core part is designed to validate the created instances of the *SecLan* conceptual model for the security DSLs and analyzers we studied. For this purpose, the survey contains questions about the correctness and completeness of the created instances capturing security DSLs *and* analyzers.

The two surveys differ in the order of questions in the second core part: the first survey asks about security DSL instances first and then about the analyzers; the second survey reverses this order. With this design, we maintain experts motivation by first asking questions about their area of expertise while also allowing them to provide feedback on both areas. Finally, we ask the experts whether they would be willing to participate in a follow-up interview to discuss the relationships calculated by *SecLan*.

The first core part contains 12 questions, the second core part contains 15 questions for security DSLs and 19 questions for analyzers. The follow-up request contains up to 2 questions. The questions for security DSLs and analyzers in the second core part can be repeated up to three times each. This repetition accounts for experts who have expertise in multiple security DSLs or analyzers (i.e., $15 * 3 = 45$ for security DSL and $19 * 3 = 57$ for analyzers). Every part can be freely skipped by the experts without providing answers to our questions. Consequently, if an expert would answer every question in both core parts for three security DSLs and three analyzers, the survey would contain in total $12 + 45 + 57 + 2 = 116$ questions. However, as we found when performing the surveys, commonly the experts only answered questions for one security DSL or analyzer.

9.1.2.1. Introduction

In the introduction of the surveys, we first explain the problem of relating security DSLs and analyzers. Then we summarize the idea of *SecLan*. Finally, we invite the experts to participate in our surveys, and present them with the repository containing the created instances of the *SecLan* conceptual model for the security DSLs and analyzers we studied (see Subsection 6.1.1).

9.1.2.2. First Core Part: Correctness and Completeness of *SecLan* conceptual model

The first core part of the surveys uses in total 12 questions to validate the correctness and completeness of the concepts and completeness of the *SecLan* conceptual model (**Q3.1**, **Q3.3**) and the correctness and completeness of the *Element Types* we provide (**Q3.2**, **Q3.4**). We first ask two questions each for the *Security Model* and the *System Model* (four questions), followed by four questions each for the *Security DSL Description* and the *Security Analyzer Description* (eight questions).

For the *Security Model*, the *Security DSL Description* and the *Security Analyzer Description*, we present the experts with the respective sub-model of the *SecLan* conceptual model and an explanation of the concepts and relations between the concepts. We ask the experts

whether "*all elements and relations in the respective sub-model fit their understanding and knowledge*", which can be answered with *yes* or *no*.

For the *System Model*, we present a description of every *Element Type*, the *Semantic Relations*, and the *If Compromised, also Compromised* relations. Note that the *System Model* presented in Subsection 6.2.2 already captures insights we gained from our surveys and interviews. The version we present to the experts in our survey can be found in our replication package [270]. Again, we ask the experts whether all "*elements and relations in the provided System Model fit their understanding and knowledge*", which can again be answered with *yes* or *no*.

For every question, if the experts answer with *yes* (i.e., they agree with elements and relations in the respective sub-model), we can conclude that the respective sub-model fits their understanding and knowledge. For every question, if the experts answer with *no* (i.e., there are concepts, *Element Types* or relations that do not fit their understanding), we ask "*which elements or relations in the respective model do not fit their understanding and why*", and provide a free-text field for their response. The free-text answers allow us to identify potential missing elements or relations, but also to determine whether the experts misunderstood the concepts and relations in the respective sub-model.

For the *Security DSL Description* and the *Security Analyzer Description*, four additional questions investigate whether their relations to the *Security Model* and the *System Model* fit the understanding of the experts. Consequently, we ask the experts "*whether all relations [from the respective sub-model] to the Security Model and System Model fit their understanding and knowledge*". These questions can also be answered with *yes* or *no*. If the experts answer the questions with *no*, we ask "*which relations [from the respective sub-model] to the Security Model or System Model do not fit their understanding and knowledge, and why*", and provide a free-text field for their response.

9.1.2.3. Second Core Part: Validating *SecLan* conceptual model Instances

The second core part of the surveys focuses on the created instances of the *SecLan* conceptual model for the security DSLs and analyzers we studied. The purpose of this part is three-fold: First, we aim to validate the correctness and completeness of the instances of the *SecLan* conceptual model to reduce threats to validity arising from our own interpretation when creating the instances. Thus, we address **Q3.5** and **Q3.6** from **G3**. Secondly, we continue to investigate completeness of the *SecLan* conceptual model on specific examples (i.e., created instances) (**Q3.3**, **Q3.4**). Thirdly, we ensure that the instances are suitable for investigating the relationships calculated by *SecLan* in the interviews.

We ask the experts 15 questions for security DSLs and 19 questions for analyzers (which can be repeated up to three times for each security DSL or analyzer the experts have expertise in). First, we ask the experts to select a security DSL or analyzer they have expertise in and the questions about the correctness and completeness of the created instance of the *SecLan* conceptual model for this security DSL or analyzer are presented to them. The four additional questions for analyzers investigate the correctness and completeness of

the weaknesses assigned to the security checks in the instances of the *SecLan* conceptual model for analyzers.

DSL or Analyzer Selection We first ask up to three questions to select the specific security DSL or analyzer and to assess their level of expertise. First, we ask the experts to select a security DSL or analyzer they have expertise in from the list of all security DSLs and analyzers we studied (see Subsection 6.1.1). This selection is essential to ensure that we can map the answers provided by the experts to the instances of the *SecLan* conceptual model correctly. To assess the level of expertise, we ask the expert whether they are main author, co-author or developer/maintainer of the respective security DSL or analyzer. If the expert selects to be a developer/maintainer, we assess how many years of experience they have with the security DSL or analyzer. After assessing the experience, we present the experts with a link to our *SecLan* repository[3] and an explanatory text of the content, so they can explore the instance they selected.

Validation of SecLan Conceptual Model Instance For each selected security DSL or analyzer, we ask the experts up to 12 questions and 16 questions respectively. These questions are designed similar to the questions regarding the correctness and completeness of the *SecLan* conceptual model in the first core part.

We first prompt the experts with the concepts and relations between the concepts used in the *Security DSL Description* or *Security Analyzer Description*, depending on whether they answer questions about either of those two. Then, we start by asking "*whether there are elements in the specific security DSL or analyzer that are not addressed by our* SecLan *conceptual model*". This question can be answered with *yes* or *no*. If the experts answer this question with yes, we ask "*which elements are not addressed by the* SecLan *conceptual model*" and provide a free-text field for their response. This question allows us to identify potential missing concepts in the *SecLan* conceptual model. Furthermore, we assess whether the experts understand the instances of the *SecLan* conceptual model created for the security DSLs and analyzers by asking up to two questions. We ask "*whether the classification of the elements in the security DSL or analyzer is clear*" which can be answered with *yes* or *no*. If the experts answer this question with *no*, we ask "*which classification is unclear and why*", again providing a free-text field for the response. This question allows us to identify potential misunderstandings of the concepts and relations in the *SecLan* conceptual model, and to rate the responses in the remaining questions accordingly.

Then, we ask up to eight questions to assess the correctness and completeness of the created instances, and the completeness of the *SecLan* conceptual model. We ask four questions for incorrectly or inappropriately classified elements and four questions for missing elements. The first question asks "*whether there are elements in the created instance that have been classified incorrectly or inappropriately*", and the second question asks "*whether there are elements missing in the created instance*". Both questions can be answered with *yes* or *no*. If

[3] `https://gravity-tool.org/seclan-repository/index.html`

the experts answer these questions with yes, we provide follow-up questions to assess *whether they can classify the incorrectly classified elements with our* SecLan *conceptual model* or *whether they can classify the missing elements with our* SecLan *conceptual model* respectively. Both questions can again be answered with *yes* or *no*.

If the experts answer one of these questions with yes, we ask a corresponding follow-up question with a free-text field for the answer:

- For incorrectly or inappropriately classified elements, we ask "*how they would classify the incorrectly or inappropriately classified elements with the* SecLan *conceptual model and why*".
- For missing elements, we ask "*how they would classify the missing elements with the* SecLan *conceptual model and why*".

The question about the rationale allows us to validate whether the expert understands the concepts of the *SecLan* conceptual model with respect to the incorrectly classified elements. Furthermore, if the expert provides a classification which fits the concepts and relations in the *SecLan* conceptual model, we can still conclude that the concepts and relations in the *SecLan* conceptual model are *correct*.

If the experts answer our questions about whether they can classify the incorrectly classified elements or the missing elements with our *SecLan* conceptual model with no, we ask a follow-up question for both aspects with a free-text field for the answer:

- For incorrectly or inappropriately classified elements, we ask "*how they would describe the class of the incorrectly or inappropriately classified elements*".
- For missing elements, we ask "*how they would describe the class of the elements missing in our classification*".

This question allows us to identify potential missing concepts in the *SecLan* conceptual model.

Correctness and Completeness of Weaknesses Assigned to Security Checks For analyzers, we ask four additional questions to investigate the correctness and completeness of the weaknesses assigned to the security checks in the instances of the *SecLan* conceptual model. For these instances, we populated the instance of the security model with the weaknesses described in the CWE [327] in version 4.12 (see Subsection 6.2.1). These questions allow us to improve the assignment of the weaknesses if necessary to increase the correctness and completeness of relationships calculated by *SecLan*.

We first ask the experts "*whether there are incorrect or inappropriate assignments of CWEs for any security check of the analyzer*", which can be answered with *yes* or *no*. If this question is answered with *yes*, we ask "*whether they can provide a correct assignment of weaknesses for the respective security checks of the analyzer*", which can be answered with a free-text field. Then, we ask "*whether there are CWEs missing for any security check of the analyzer*", which can be answered with *yes* or *no*. If the question is answered with *yes*,

we ask them "*to provide us with the missing CWEs for the respective security checks of the analyzer*", which can be answered with a free-text field.

9.1.2.4. Follow-up Interview Request

At the end of the survey, we ask every expert whether they would agree to be contacted for a follow-up interview to investigate the relationships calculated by *SecLan* between security DSLs and analyzers of their choice. For this purpose, we provide two questions in the survey. The first question asks "*whether the expert would be open to participating in an interactive follow-up interview*", which can be answered with *yes* or *no*. If this question is answered with yes, we ask the experts for their contact information, including (1) their *Name and Surname*, (2) their *Email Address*, and (3) *the preferred security DSL or Analyzer* they would like to investigate in the interview.

9.1.3. Interview Design

The interviews primarily aim to qualitatively explore the relationships calculated by *SecLan* between security DSLs and analyzers (**G4**). Before exploring these relationships, we also validate the correctness and completeness of the *SecLan* conceptual model instances with each expert (**G3**), ensuring that subsequent relationship assessments are based on accurate representations. This validation step is crucial as we modify the instances of the *SecLan* conceptual model based on the feedback provided by the experts in the surveys (see Subsection 9.1.2).

We design semi-structured interviews to achieve both objectives. This format is particularly suitable for assessing the correctness of the relationships, as experts in security DSLs know what can violate the security described in the DSLs. Conversely, experts in analyzers are aware of which vulnerabilities they find and which security aspects the detected vulnerabilities can violate. Semi-structured interviews also allow experts to raise concerns about aspects of the calculated relationships we do not anticipate, such as practical applicability or scalability of the approach.

We aim to minimize effort for the experts to motivate them to participate in the interviews. Thus, the interviews will be held over the video conference tools Microsoft Teams or Zoom to reduce the obstacles to participation for the experts. Furthermore, we limit the duration of the interviews to one hour to avoid high time investment for the experts and to limit fatigue effects. Limiting the duration of the interviews also allows us to schedule the interviews more easily with the experts because they can more easily determine a suitable time slot. One collaborator conducted all interviews to ensure consistency in the interview process, while other collaborators wrote a protocol. We do not record the interviews to lower the barrier to participation and reduce privacy concerns. In addition, we also ensure the anonymity of the experts when reporting results.

9.1.3.1. Interview Preparation

We contact all experts who agree to the follow-up interview in the survey (see Subsection 9.1.2). We adapt the instances of the *SecLan* conceptual model to address the feedback of the experts from the surveys regarding the correctness and completeness of the created instances of the *SecLan* conceptual model for the security DSLs and analyzers we studied. We only modify the created instances if the experts identify incorrect classifications or missing elements that they could describe using the concepts and relations in the *SecLan* conceptual model. These modifications have to be completed before conducting the interviews to ensure all experts evaluated consistent instances. With this approach, we ensure that the created instances of the *SecLan* conceptual model for the security DSLs and analyzers are correct and complete according to the understanding of the experts.

As we performed remote interviews, we prepared presentation slides that introduce the topic, present the *SecLan* conceptual model, the *System Model* (*Element Types* and the relations between the *Element Types*), and an example instance of the *Security Model*.

9.1.3.2. Interview Structure

We design the semi-structured interview to include five parts: **Introduction**, **Consistency**, **Instance Correctness**, **Relationships**, and **Closing**. Since the parts **Consistency** and **Instance Correctness** share a common goal — verifying the correctness and completeness of the *SecLan* conceptual model and the instance under investigation in the interview — we discuss them together below.

Introduction In the first part (**Introduction**), we introduce the topic, present an example scenario, and explain the purpose of *SecLan* and the *SecLan* conceptual model to experts. We also ask whether experts know related approaches connecting design-time security DSLs to implementation-level security analysis, and we collect any questions they have before beginning the core interview parts. This introduction allows experts to refresh the understanding of the topic before we begin exploring the relationships calculated by *SecLan*.

Consistency & Instance Correctness The second and third part (**Consistency** and **Instance Correctness**) aim to validate the *SecLan* conceptual model and its instances before exploring relationships. We discuss them together because they share this common goal.

Consistency. In the second part we investigate whether the concepts and relations between the concepts of the *SecLan* conceptual model still fit the understanding of the experts. This allows us to verify their survey responses. Here, we gather insights about the correctness and completeness of the *SecLan* conceptual model which are not mentioned in the survey. We discuss any concerns raised by the experts to ensure that we have a common understanding of the concepts and relations in the *SecLan* conceptual model before we continue with the next part. In this part, we collect information and feedback

but do not modify the *SecLan* conceptual model, unless experts identify fundamental inconsistencies that require changes.

Instance Correctness. The third part verifies the correctness and completeness of the instances that we adapt based on survey feedback (see Subsection 9.1.2). We discuss concerns of the experts about their selected *SecLan* conceptual model instance and, if necessary, make minor corrections based on their feedback. This validation ensures we understand the survey feedback correctly and adapt the instances accordingly. Without correct instances, the validity of the assessments of the relationships calculated by *SecLan* would be compromised. Additionally, this part ensures experts understand their instances before we discuss calculated relationships.

If modifications are made during this part, we recalculate the relationships using *SecLan* based on the updated instance. Given our one-hour time limit, we aim to complete this part as quickly as possible. We immediately continue with the next part if experts raise no concerns.

Relationships In the fourth part (**Relationships**), we investigate the relationships calculated by *SecLan* between the selected security DSL and all security checks from the studied analyzers or vice versa. We show the relationships derived from the instance of the *SecLan* conceptual model for the selected security DSL and the security checks of the created instances for all studied analyzers. We explore these relationships for approximately 40 minutes (depending on the time taken for the prior three parts), concluding about 10 minutes before the end to allow time for closing questions and feedback. During the interview, we accept all feedback experts provide about the relationships calculated by *SecLan*. For example, as we will see later in Section 9.2, the experts provided valuable feedback about the number of relationships calculated by *SecLan* which was not the focus of the interview.

First, we show all suggested security checks and their analyzers or the suggested security DSLs and their security aspects (depending on whether the expert selected an security DSL or analyzer). If the expert selected an security DSL, we show all security checks and their analyzers which are calculated to be related to the selected security DSL. Based on this information, we ask the experts "*whether they know any of them*". If the experts answer with *yes*, we ask them to select the known checks or security DSLs from the list of all suggested checks or security DSLs. For known checks, we ask "*whether they think that any of the suggested checks they know work to detect vulnerabilities that violate a security aspect described in the selected security DSL*". If the expert selected an analyzer, we ask the same questions for the suggested security DSLs and their security aspects. These questions allow us to filter the relationships which the expert can provide feedback on with a high degree of confidence.

After this filtering, we present relationships iteratively and ask experts *whether they agree with the relationships* (i.e., whether vulnerabilities detected by the check violate security properties described in their security DSL or vice versa). For any disagreement, we review the reasoning which *SecLan* provides by using the paths calculated between the *Security*

Check and *Security Aspect* through the *Security Model* (threats and weaknesses) and the *System Model* (*Element Types*). We discuss each part of the path with the expert, such as the weaknesses and threats involved in the relationship or the propagation over the *Element Types*. We also investigate the documentation of the security check (if available) and the descriptions of the weaknesses in the CWE database. Based on this discussion, we ask whether the experts are still of the opinion that the findings of the security check do not violate security properties described in their security DSL. This approach allows us to gather insights whether the reasoning can help experts to discover relationships between security checks and security DSLs which they would not have found without *SecLan*.

Closing In the final part (**Closing**), we ask experts whether they have any remaining questions or remarks. We also inquire whether "*they think the* SecLan *conceptual model fits its purpose, whether they have suggestions for improvement*", and "*whether* SecLan *and the* SecLan *conceptual model could find application in practice*". Finally, we thank experts for their time and valuable feedback.

9.1.4. Calculating Inter-Rater Reliability from Survey Results

We now detail how we calculate our inter-rater reliability metrics from the survey results. For every question in the survey, we collect answers about the correctness of the concepts and relations between the concepts in the *SecLan* conceptual model as well as about the correctness of the *Element Types* and relations between the *Element Types*. We encode every concept and relation between the concepts or *Element Types* and relations between the *Element Types* into a column of an array. We encode every expert's answer in the respective row of the array.

In our survey, we ask the experts for every sub-model of the *SecLan* conceptual model, whether *all elements and relations in the respective sub-model fit their understanding and knowledge*. We mark each concept and relation (or *Element Types* and relations between them in case of the *System Model*) in the respective sub-model as agreed if an expert answers *yes*; encoding them with 1 in the respective cells. If the experts answer *no*, we mark concepts and relations between the concepts mentioned in the free-text field for the follow-up question *which elements or relations in the respective model does not fit their understanding and why* as disagreements; encoding them with 0 in the respective cells. If the experts do not answer a question regarding a sub-model (i.e., N/A for a sub-model), we remove this expert's row for the respective sub-model from the array.

In the investigation of the instances created with the *SecLan* conceptual model, the experts can indicate whether there are *inappropriately or incorrectly classified elements* for which they cannot describe a correct classification with the *SecLan* conceptual model. We only modify the prior assessments of the concepts and relations between the concepts in the *SecLan* conceptual model if the experts cannot provide a correct classification and explicitly describes that a concept or relation does have incorrect semantics to be applicable.

```
install.packages("irr")
install.packages("irrCAC")
library(irr)
library(irrCAC)

responses <- rbind(\dots) # provide matrix with survey responses
alpha <-  kripp.alpha(responses, method="nominal")$value # Calculate Krippendorff's alpha
pa <- agree(responses)$value  # Calculate Percentage Agreement
ac1 <- gwet.ac1.raw(responses)$est$coeff.val  # Calculate Gwet's AC1
```

Listing 9.1: Commands and sequence in R to calculate inter-rater reliability metrics from survey results

With this encoding, we apply the statistics tool R [59] to calculate the inter-rater reliability metrics Krippendorff's α [175], Percentage Agreement PA, and Gwet's $AC1$ [110] for every sub-model of the *SecLan* conceptual model. Listing 9.1 shows the commands we use in R to calculate the inter-rater reliability metrics. For calculating Krippendorff's α, we use the `kripp.alpha` function from the `irr` package. For calculating the Percentage Agreement, we use the `agree` function from the `irr` package. For calculating Gwet's $AC1$, we use the `gwet.ac1.raw` function from the `irrCAC` package. The Percentage Agreement requires the columns and rows in the arrays to be inverted to the encoding we presented before. Thus, we transpose the array before calculating the Percentage Agreement. Our replication package [270] includes the files containing the encoded survey results as well as R scripts to calculate the inter-rater reliability metrics.

9.2. Evaluation Results and Discussion

This section presents the results of our evaluation of *SecLan* (**C3**). We start with presenting and discussing the results regarding the *correctness and completeness* of the *SecLan* conceptual model (**G3**). Then we present the results of the qualitative exploration of the relationships calculated by *SecLan* (**G4**).

Table 9.1 shows the experts of the surveys and interviews, the analyzer or DSL they are associated with, their role, and their expertise with the respective DSL or analyzer. Furthermore, this table shows which experts participated in the survey and the qualitative exploration of the relationships in the interviews.

We contacted authors and developers/maintainers of all 66 security DSLs and 33 source code analyses we studied (see Subsection 9.1.1) by email. Of all contacted experts, 17 participated in our survey and 7 participated in interviews. From the contacted authors of scientific publications about security DSLs and analyzers, 13 agreed to participate in our survey (11 security DSLs and 2 analyzers) and 6 participated in interviews (all experts in security DSLs). Of all contacted developers/maintainers *which are not also authors of scientific publications about security DSLs and analyzers*, 3 agreed to participate in our survey (2 security DSLs and 1 analyzer) and one participated in the interviews (expert in

a security DSL). One survey performed did not select any security DSL or analyzer and also did specify its role or expertise; we still include this survey in our results because it answers questions of the first core part. The experts required between 30 minutes and one hour to complete the surveys.

9.2.1. Result Presentation and Interpretation: Correctness and Completeness of the *SecLan* conceptual model and its Instances

Here we report and interpret our results to answer **Q3.1**, **Q3.2**, **Q3.3**, **Q3.4**, **Q3.5** and **Q3.6** regarding the correctness and completeness of the *SecLan* conceptual model. The results are based on the instances of the *SecLan* conceptual model we created for the 66 examined security DSLs and the 33 analyzers with their 408 security checks as well as the responses of the 17 experts that participated in our survey.

For structuring the presentation of results, we partition the results similarly as the discussed questions in Section 7.1. We first present the results regarding the correctness and completeness of the concepts and relations between the concepts in the *SecLan* conceptual model as well as the instances of *Element Types* and relations between them (**Q3.1**, **Q3.2**, **Q3.3** and **Q3.4**). Then we present the results regarding the correctness and completeness of the created instances of security DSLs and source code analyses considered in our study (**Q3.5** and **Q3.6**).

9.2.1.1. Results for Correctness of the *SecLan* conceptual model Concepts and Relations

We received 17 responses to the first core part of our survey that addresses the *correctness* of the concepts and relations between the concepts in the *SecLan* conceptual model (**Q3.1**) and the specific *Element Types* and the relation between these *Element Types* (**Q3.2**). Based on these answers we present the inter-rater agreement metrics Krippendorff's α (**M3.1.1**), percentage agreement *PA* (**M3.1.2**), and Gwet's *AC*1 (**M3.1.3**). We report the results for every sub-model of the *SecLan* conceptual model separately and summarize them in Table 9.2.

Security Model All 17 experts provided feedback regarding the *Security Model*. 16 of the 17 experts in the survey agreed with the concept *Security Objectives*, *Threat* and the *Weakness* as well as the relations between these concepts. One expert expressed a concern regarding the relation between *Threat* and *Weakness*, by stating that a threat can exist independent of a weakness in the system. We do not object to this statement, but it does not contradict the *SecLan* conceptual model which describes only that a *Weakness* enables at least one threat and not vice versa. Still, to account for this feedback, we count a *disagreement* regarding the relationship between *Threat* and *Weakness*.

ID	Analyzer or DSL Name	Type	Role	Expertise (in years)	Participation					
					Valid.					Expl.
					Sec. Mod.	Sys. Mod.	DSL Desc.	Ana. Desc.	Application	
I1	SecBPMN	DSL	MA, D/M	> 5	✓	✓	✓	✓	✓	✓
I2	SecDFD	DSL	MA, D/M	> 5	✓	✓	✓	✓	✓	✓
I3	ADF	DSL	MA, D/M	3 – 5	✓	✓	✓	✓	✓	✓
I4	UMLsec	DSL	CA	> 5	✓	✓	✓	✓	✓	✓
I5	UMLsec	DSL	D/M	1 – 2	✓	✓	✓	✓	✓	✓
I6	Att. Prop.	DSL	MA, D/M	3 – 5	✓	✓	✓	✓	✓	✓
I7	PbSD	DSL	CA	n/a	✓	✓	✓	✓	✓	✓
I8	CARDS	DSL	CA, D/M	< 1	✓	✓	✓	✓	✓	×
I9	SysML-Sec	DSL	MA	n/a	✓	✓	✓	✓	✓	×
I10	AFT	DSL	MA	> 5	✓	✓	✓	✓	✓	×
I11	PMD	Analyzer	D/M	> 5	✓	✓	✓	✓	✓	×
I12	SecureUML	DSL	MA	n/a	✓	✓	✓	✓	✓	×
I13	CppCheck	Analyzer	MA, D/M	> 5	✓	✓	×	×	×	×
I14	n/a	Analyzer	n/a	n/a	✓	✓	×	✓	×	×
I15	FlowDroid	Analyzer	MA, D/M	> 5	✓	✓	✓	✓	✓	×
I16	CCMsec	DSL	CA	n/a	✓	✓	✓	✓	×	×
I17	UMLsec	DSL	D/M	3 – 5	✓	✓	✓	✓	✓	×

Table 9.1.: Experts in the validation and qualitative exploration. **ID** provides an identifier and the assigned Analyzer or DSL (column **Analyzer or DSL Name**). Some DSL or analyzer names abbreviated: Architectural Data Flow DSL [300] (*ADF*), Attack Propagation DSL [348] (*Att. Prop.*) and Attack-Fault Trees [180] (*AFT*). The order of the experts is not chronological; experts who participated in the exploration are shown first. The column **Role** indicates the roles of the experts in the development of the respective security DSL or analyzer: Main Author (*MA*), Co-Author (*CA*), and Developer or Maintainer (D/M). The columns in the column Participation indicate which parts of the survey and interview the experts participated in. The column **Expertise** shows the experts' expertise with the respective security DSL or analyzer *in years*. The column **Participation** indicates which parts of the survey and interview the experts participated in. **Valid.** indicates in which parts the experts participated in the survey to validate the sub-models of the *SecLan* conceptual model (Security Model (*Sec. Mod.*), System Model (*Sys. Mod.*), Security DSL Description (*DSL Desc.*), Security Analyzer Description (*Ana. Desc.*)) and its application (column *Application*) to the security DSLs and analyzers. **Expl.** indicates whether the expert participated in the qualitative exploration in the interviews.

Sub-model	#Experts	#Elements	#Disagr.	PA	α	$AC1$
Security Model	17	6	1	0.833	-0.009	0.980
System Model	17	33	2	0.939	-0.003	0.992
Security DSL Descr.	15	8	3	0.875	0.121	0.954
Security Analyzer Descr.	16	5	1	0.800	-0.011	0.974

Table 9.2.: Summary of inter-rater agreement metrics for each sub-model of the *SecLan* conceptual model: the Security Model, the System Model, the Security DSL Description (Descr.), and the Security Analyzer Description (Descr.). Also showing how many experts provided feedback for the given sub-model (#Experts), the total number of concepts and relations between the concepts (or *Element Types* and relations between them for the *System Model*) captured in the respective sub-model (#Elements) and the number of disagreements counted based on the survey responses (#Disagr.). The summary also includes the calculated inter-rater agreement metrics Percentage Agreement (PA), Krippendorff's α (α) and Gwet's $AC1$ ($AC1$).

In total, we count *one disagreement* for the concepts and relations between the concepts in the *Security Model*. This feedback results in $PA = 0.833$, Krippendorff's $\alpha = -0.009$ and Gwet's $AC1 = 0.980$.

System Model All 17 experts provided feedback regarding the specific *System Model*. 15 of the 17 experts fully agreed with the specific *Element Types* and the relations between the *Element Types* we provide. Two experts raised minor concerns regarding the *semantic relations*. One expert expressed concerns about the relation between *Information Flow* and *Entity*, because an information leak may not only occur due to a single *Entity* (we already modified this relation in our final *SecLan* conceptual model illustrated in Figure 6.4 as discussed later in this section). Despite this concern, the expert agreed in the interview that this *Entity* is what you want to have to pinpoint the locations to fix information flow violations. This expert statement confirms the fact that there is a relation between *Information Flow* and *Entity* to be considered. The other expert confirmed the semantic relation between *Control Flow* and *Information Flow* but expressed that this relation could also be realized transitively over *Activity*. In the interview, the expert agreed that the direct relation between *Control Flow* and *Information Flow* is valid due to the definition of *Information Flow* in literature [20]. Consequently, we maintained this relation in the *SecLan* conceptual model. Although both experts raised concerns, they agreed that these concerns do not invalidate the concepts or relations in the *System Model*.

Still, we use a conservative approach and count *two disagreements* regarding the *System Model* in total. This feedback results in $PA = 0.939$, Krippendorff's $\alpha = -0.003$ and Gwet's $AC1 = 0.992$ for the specific *Element Types* and relations provided in the *System Model*.

Security DSL Description 15 of the 17 experts provided feedback regarding the *Security DSL Description*. All experts fully agreed with the concepts *Security DSL*, the *Specification*

Element and the relationships between the *Security DSL Description* and the *System Model*. However, three experts expressed concerns regarding *Security Aspect*. The first expert stated that the goal model community defines that *Confidentiality* and *Integrity* should be enforced by a security DSL and, consequently, are *Security Aspects*. In contrast, we subsumed them explicitly under the concept *Security Objective* in the *Security Model* and create a relation through the *Threat* concept. This expert agreed in the interview to this modelling approach as a valid indirection. The second expert asked why a *Security Aspect* cannot immediately remedy a *Weakness*. We address this relation indirectly via *Threats* that enable *Weaknesses*. We see this approach as valid in the context of *SecLan* because the relationships between security DSLs and analyzers are established through the same path. The third expert mentioned that "Security Aspect sounds pretty generic" and that "security control or counter measure is the term typically used"; thus disagreeing with the name of the concept.

In total, we count *three disagreements* regarding the *Security DSL Description*: one disagreement regarding name of *Security Aspect*, and two disagreements regarding the relations between the *Security Aspect* and the *Security Model*. This feedback results in $PA = 0.875$, Krippendorff's $\alpha = 0.121$ and Gwet's $AC1 = 0.954$ for the *Security DSL Description*.

Security Analyzer Description 16 of the 17 experts provided feedback regarding the *Security Analyzer Description*. From these experts, 15 fully agreed with the concepts *Static Analyzer*, *Security Check* and the relations between these concepts and the *Security Model* as well as the *System Model*. One expert expressed concerns regarding the relation between the *Security Check* and *Element Type*; stating that an analyzer does not analyze an *Element Type* but rather elements of a *specific type*. We could not clarify this feedback in an interview; thus we count this as disagreement regarding the relation between *Security Check* and *Element Type*.

In total, we count *one disagreement* regarding the *Security Analyzer Description*. This feedback results in $PA = 0.800$, Krippendorff's $\alpha = -0.011$ and Gwet's $AC1 = 0.974$ for the *Security Analyzer Description*.

9.2.1.2. Results for Correctness and Completeness of the Created Instances of security DSLs and Analyzers

We received 14 responses to the second core part of our survey, covering 10 security DSLs (we received three responses for one DSL) and two analyzers. 12 responses were from experts in security DSLs and two responses were from experts in analyzers. The security DSLs we received responses for are CARDS [150], Attack-Fault Trees [180], PbSD [2], SecBPMN [288], SecDFD [338], SecureUML [192], SysML-Sec [17], UMLsec [154] (in three responses), the Architectural Data Flow DSL [300] and the Attack Propagation DSL [348]. These security DSLs cover a wide range of system descriptions, including business process models, data flow diagrams, component-based architectures and UML diagrams. Security aspects addressed are confidentiality, integrity, authenticity of data,

accountability, auditability, secure data flow, access control, secure dependencies, and communication security. The analyzers we received responses for were the *single-purpose* information flow analyzer FlowDroid [19] and the *multi-purpose analyzer* PMD [260].

Seven experts provided feedback that we did not classify elements of their security DSLs or classified them incorrectly. One expert mentioned that *counter measures* in their security DSL cannot be expressed with the *SecLan* conceptual model. Furthermore, this expert mentioned that the elements *Analysis*, *Interaction Points*, *Consistency Between Views* and *Inter-relations with safety and performance* cannot be classified with the *SecLan* conceptual model. Another expert mentioned that we misclassified the elements *AssemblyContext* or *ResourceContainer* in the Attack Propagation DSL wrongly as specification elements. The expert stated that the elements *EntityMatch* and *MethodMatch* are only wrappers for the elements of the PCM (the architecture description language used in the security DSL). We agree with this feedback and removed these elements from the instance of this security DSL. The same expert also points out that it is unclear how to express the *indirectly derived mitigations* in their analysis. These mitigations are similar to our *Security Aspects* but are derived from the system structure and not explicitly modeled. Another expert mentioned that "PC (Protection Constraint) is quite abstract. It is not clear how constraints can be specified. Perhaps through the specification element" (the element in the security DSL). Furthermore, the expert mentioned that the specification element *Integrity* of SecBPMN has two semantics: to label security requirements of *Data* (as we considered it) and to express integrity requirements of a task. Therefore, we included a reference between this *Specification Element* and the *Element Type Activity*. Another expert stated that a reference between a *Specification Element Trust Zone* and the *Element Type Information Flow* should be included in SecDFD. Another expert removed the reference of the *Specification Element Usage Scenario Characterization* to *Element Type Information Flow* in the architectural data flow security DSL. One expert argued that *Specification Elements* of the *Security Aspect Secure Dependency* in UMLSec should only reference *Information Flow*, while we also referenced *Control Flow*. The second expert in UMLSec did agree with the prior modeling without expressing concerns.

One expert provided feedback about the analyzer *FlowDroid* regarding elements that we did not classify correctly with the *SecLan* conceptual model. The expert states that the analysis *does not really fit our model* because it is intended to be used *as a building block for many types of static vulnerability analysis*. The concern of the expert is that *our model seems to be focused more on tools that look for a specific type of vulnerability*. Furthermore, the expert mentioned that in *FlowDroid there is no information on the system model. There are no element types.*

We also received feedback about the *weaknesses* that we modeled for specific analyzers. One expert mentioned that *FlowDroid* "can also be used to detect injection vulnerabilities, e.g., CWE-89". The same expert highlights, that *FlowDroid* "can build many different security analysis on top of FlowDroid and that these tools can then look for many different CWEs". We also received feedback regarding *PMD* where we assigned *CWE-327: Use of a Broken or Risky Cryptographic Algorithm* to *PMD*'s check *HardCodedCryptoKey*. The expert mentioned that "the CWEs listed are surely related to the same issue, but their security

checks are usually very, very simple." Especially that "CWE-327 does not really fit for this check because it does not check for specific algorithms but only for hard-coded values". The expert also provided the test cases for this check for us to validate. We agree with this incorrect assignment and removed it from our instances.

9.2.1.3. Interpretation of Results for Correctness and Completeness of the *SecLan* Conceptual Model

We now discuss the results obtained for the correctness and completeness of the *SecLan* conceptual model regarding the concepts and relations between the concepts as well as the *Element Types* and relations between them. From the 17 experts that participated in our survey, we received a total of seven *disagreements* regarding all concepts and relations between the concepts as well as the provided *Element Types* and relations between the *Element Types*.

We want to highlight some special cases of expert feedback we received. In the *Security Model*, one expert expressed that *Threats* can exist independent of *Weaknesses*. While we calculated a disagreement for this feedback, we want to highlight that our *SecLan* conceptual model does not contradict this statement. The *SecLan* conceptual model only states that a *Weakness* enables at least one *Threat*. However, the direction of the relation does not indicate that the *Threat* cannot exist without a *Weakness*.

In the *System Model*, two experts agreed that the *Element Types* and relations we provided are valid despite having minor concerns. One expert expressed concerns regarding the relation between *Information Flow* and *Entity*, stating that an information leak may not only occur due to a single *Entity*. However, the expert agreed in the interview that this *Entity* is what you want to have to pinpoint the locations to fix information flow violations. Still, we already contain an implicit relation between *Information Flow* and *Entity* through *Data* and *State*. We account for the concern of the expert by (1) removing the direct semantic relation between *Information Flow* and *Entity* in the *SecLan* conceptual model, and (2) establishing a *If Compromised, also Compromised* path transitively between *Information Flow*, *Data*, *State* and *Entity*. This reflects the expert feedback as data and state can be shared between entities, while still maintaining the capability to relate *Information Flow* to *Entity*. Still, already in the initial version of the *SecLan* conceptual model, the expert stated that this relation is necessary to have. As the second expert agreed about our representation of *Information Flow* to have a relation with control flow, we maintained this relation in the *SecLan* conceptual model despite counting a disagreement for this feedback.

For the *Security DSL Description*, we received feedback regarding the concept *Security Aspect* that it should more likely be named *Security Control* or *Countermeasure*. However, the expert did not object to the concept itself but rather to the name.

FOr the *Security Analyzer Description*, expert stated concerns regarding the relation between *Security Check* and *Element Type* in the *System Model* because *an analyzer does not analyze an Element Type but rather elements of a specific type.* Although we agree with this statement, the specific *System Model* of the *SecLan* conceptual model abstracts such

specific types exactly with the *Element Types* provided. Consequently, we interpret this as a terminology mismatch between the expert and our concept of *Element Type* applied in the *SecLan* conceptual model. As stated before, we see this as a misunderstanding of our *SecLan* conceptual model rather than an actual disagreement with the concept of *Element Type*.

To establish a relation between the disagreements and the *SecLan* conceptual model, we count all concepts, *Element Types* and relations we define. The *Security Model* contains 3 concepts and 3 relations. The *Security DSL Description* contains 3 concepts and 5 relations. The *Security Analyzer Description* contains 2 concepts and 3 relations. In addition, the *System Model* we provided to the experts contains 9 *Element Types*, 14 *Semantic Relations* and 10 *if compromised, also compromised* relations. Thus, every expert could disagree with a total of 52 elements in the *SecLan* conceptual model and the *Element Types* and relations between the *Element Types* provided. Considering sub-models for which an expert provided no feedback at all, we could receive a total of 102(Security Model) + 120(Security DSL Description)+80(Security Analyzer Description)+561(System Model) = 863 disagreements or agreements.

We received a total of seven disagreements, resulting in a ratio of disagreements to total possible disagreements of $\frac{7}{863} = 0.008$. The ratio can be seen as small, indicating that the *SecLan* conceptual model is correct in the concepts, provided *Element Types* and relations it contains. This is also shown by the high values of percentage agreement PA (range between 0.800 and 0.939) as well as Gwet's $AC1$ (range between 0.954 and 0.992). Krippendorff's α shows low values (ranges between -0.011 and 0.121) which contrasts the high values of PA and Gwet's $AC1$. The values of Krippendorff's α arise due to the assumption that valid responses are not homogeneous (i.e., highly homogeneous responses are penalized; see our discussion in Section 7.1). However, in our case, we see that the responses were highly homogeneous (i.e., most experts agreed with the *SecLan* conceptual model). Thus, we see the high values of PA and Gwet's $AC1$ as a better representation of the inter-rater agreement in our case and conclude that the *SecLan* conceptual model is correct regarding the concepts and relations between the concepts as well as the *Element Types* and relations between them. Overall, we conclude that the concepts and relation between the concepts in the *SecLan* conceptual model as well as the *Element Types* and relations between them are correct based on the expert feedback we received. Thus, we answer **Q3.1** and **Q3.2** with yes; the concepts and relations between the concepts in the *SecLan* conceptual model as well as the *Element Types* and relations between them are correct.

14 experts provided feedback for 10 instances of the *SecLan* conceptual model for specific security DSLs and analyzers. Four of these experts provided feedback about the classifications of elements that were missing. One expert mentioned that we missed *constraint policies* but that they could be expressed through *Specification Element*. Thus, we do not count this as a missing concept in the *SecLan* conceptual model, as the expert agreed that it can be expressed through an existing concept. However, two experts mentioned that concepts in their security DSLs could not be expressed with the *SecLan* conceptual model. One expert detailed that *counter measures* in their security DSL cannot be expressed. We did not include counter measures explicitly in the *SecLan* conceptual model because only

some security DSLs provide explicit counter measures while others only provide security requirements or constraints. Thus, we subsume counter measures as a combination of *Security Aspect* and *Specification Element*. Still, we count this as a missing concept in the *SecLan* conceptual model and plan to investigate whether we should include *Counter Measure* as a separate concept in future work (see Section 11.3).

Regarding the analyzers, one expert mentioned that *FlowDroid* can be used as a building block for many types of static vulnerability analyses and thus does not fit well in our model. Although we agree with this statement, Peldszus et al. [252] also use it to detect information flow violations directly. Thus, we see this feedback as a misunderstanding of our *SecLan* conceptual model rather than an invalidation. We received feedback regarding incorrect or inappropriate assignments of *Weaknesses* to specific analyzers. However, this feedback was minor because we only had to remove incorrect assignments or add missing ones. Overall, we conclude based on the feedback about the instances of the *SecLan* conceptual model for specific security DSLs and analyzers that the *SecLan* conceptual model is mostly complete regarding the concepts it provides to classify elements in security DSLs and analyzers. Including concepts such as *counter measures* could improve the completeness of the *SecLan* conceptual model further. However, we have to investigate whether including such concepts would improve capability of *SecLan* to detect relationships between security DSLs and analyzers. Considering the feedback about missing concepts, we answer **Q3.3** and **Q3.4** with yes; the *SecLan* conceptual model is mostly complete regarding the concepts it provides to classify elements in security DSLs and analyzers.

In the feedback, we found two main challenges regarding the correctness and completeness of the created instances of the *SecLan* conceptual model for specific security DSLs and analyzers. The expert providing feedback about the security DSL of Walter et al. [348] expressed that we misclassified elements in their security DSL which are hard to be classified with the *SecLan* conceptual model. In this security DSL, we had difficulties to incorporate *indirectly derived mitigations* that are not explicitly modeled in the security DSL. An additional challenge in this approach arises that it is similar to *SecLan* in the sense that it derives attack paths and mitigations based on the system structure, vulnerabilities annotated to it and an access control model. Thus, the vulnerabilities we aim to detect with *SecLan* are already incorporated in the security DSL. We see this as a limitation of our approach because the purpose of our *SecLan* conceptual model is to relate security DSLs and analyzers by discovering these vulnerabilities explicitly.

All other experts did not express major errors in the created instances of the *SecLan* conceptual model for specific security DSLs and analyzers that would invalidate our created instances. However, some experts requested changes of individual elements. The most notable change arose in SecBPMN [288] because of the ambiguous semantics of *Integrity*. We misinterpreted the semantics of *Integrity* to only label security requirements of *Data*, while the expert highlighted that it can also express integrity requirements of a *Task*. We resolved this issue with our *SecLan* conceptual model based on the experts' feedback by adding additional relation reflecting both semantics.

In total, we had to make minor adjustments to the other security DSLs and analyzers based on the expert feedback, which included adding or deleting references between *Specification*

Elements and *Element Types* or removing incorrect assignments of *Weaknesses* to analyzers. Thus, asides from the security DSL of Walter et al. [348], we answer **Q3.5** and **Q3.6** with yes; the *SecLan* conceptual model is capable of describing security DSLs and analyzers.

As we answered all questions **Q3.1**, **Q3.2**, **Q3.3**, **Q3.4**, **Q3.5**, and **Q3.6** positively, we conclude that we achieved **G3**, namely the *SecLan* conceptual model and the instances we created are correct and complete.

9.2.2. Result Presentation and Interpretation: Qualitative Exploration of Relationships

In this section, we present the results of our qualitative exploration of the relationships calculated by *SecLan* between security aspects of security DSLs and security checks of source code analyzers. We provide the results of the interviews with the seven experts that participated in the qualitative exploration in the following. In contrast to the interpretation of questions stated in the GQM plan (see Section 7.1) we do not answer predefined questions but present the insights we gathered from the interviews. Still, we consider these insights to be an interpretation of our evaluation results regarding the qualitative exploration of the relationships calculated by *SecLan* as we do not claim to have exhaustive coverage of all possible relationships.

9.2.2.1. Results of the Qualitative Exploration

Seven experts participated in the qualitative exploration of the relationships calculated by *SecLan* after participating in the survey. All interviewees were experts in security DSLs; namely SecBPMN [288] (I_1), SecDFD [338] (I_2), the Architectural Dataflow DSL [300] (I_3), UMLsec [154] (I_4),(I_5), the Attack Propagation DSL [348] (I_6), and PbSD [2] (I_7). All experts interviewed held a Ph.D. or were Ph.D. students. Two interviewees went to industry after gaining a Ph.D., while the other five interviewees are working in academia (three faculty positions, one postdoctoral researcher, and one research associate). We calculated 38,136 relationships between security aspects of the 66 security DSLs and the 408 security checks of the 33 source code analyzers we studied. This amount arises due to the broad scope of security DSLs. The extent of this amount got evident during the interviews, as the experts could not inspect all relationships in detail within the time frame of the interviews. Still, all experts together reviewed hundreds of relationships.

During the interviews, we identified six false positives, all of which were reported in I_1 and I_3. The false positives all involved security checks of *SonarQube* [313]. When investigating the false positives, I_3 stated that *these relationships are most likely arising because the checks are pointing to general weaknesses such as CWE-20: Improper Input Validation.*

In many interviews, we observed the scenario that the expert initially flagged some security checks as false positives but then acknowledged a relationship or potential relationship

with the security DSLs after investigating the rationale provided by *SecLan*. In this process, the experts investigated the reasoning (i.e., the path calculated by *SecLan* over the instances of the *SecLan* conceptual model) between security aspect and the initially flagged security checks in detail. The experts looked into the description of the CWEs and the security checks to which *SecLan* provided pointers to. Especially experts in I_2, I_3, I_4 and I_6 acknowledge after this investigation that the security check was or could be related to the security aspect.

The benefits of the relationships calculated by *SecLan* were then highlighted by several experts. In interviews I_1 and I_3, the experts acknowledged that most of the relationships are clear. For example, the expert in I_3 stated *most of the things [checks] recommended are pretty clear.* The interviews I_1, I_2, I_3, I_4 and I_6 showed that the paths provided by *SecLan* helped the experts to understand the relationship between the security DSL and the security check. The experts in the interviews I_2, I_3 and I_4 explicitly stated that the relationship became particularly clear when they more closely examined the CWEs and security check descriptions referenced by *SecLan*. For example, the expert in I_2 was not sure *how it [SonarQube rule RSPEC-5679] is related to secDFD*. Then the expert investigated the security check description *and* the path provided by *SecLan* (especially the contained weaknesses *CWE-287* and *CWE-347*). Considering both information, the expert stated that *first looking at the [check] description, it was hard to understand, but looking at the explanation [the relation], it's clear.* In I_3, the expert stated that the relation is not always immediately clear, *but after looking it [the relation and descriptions of the weaknesses] up, it gets clear why it is related.* In I_4, the expert inspected *SonarQube rule RSPEC-1989* and stated that *after looking into the reasoning and check description, the check is related.*

Another topic which often arose was the amount of relationships calculated by *SecLan*. Four experts (I_1, I_2, I_3 and I_6) stated that reduction of the number of relationships would be beneficial. The expert in I_1 *was wondering whether some security relations can be removed.* This expert suggested research into prioritization, for example, by *considering the level of propagation, such as level zero (no step), level one (one step)…* The expert in I_2 stated that *the number of warnings [relations] should be reduced as everything is connected right now.* The expert in I_3 highlighted that this amount of relationships could be caused *by the CWEs assigned to the security checks being too general or to be aimed at too broad threats such as Information Disclosure.* The expert in I_5 raised the concerns that *the number of checks recommended for secure links [security aspect in UMLSec] is a lot*, but that *less recommended checks would be ok.* The expert in I_6 stated that *all means that surround the [access control] engine […] is a threat to access control.* Consequently, anything detected by the checks that could lead to bypassing an access control engine running PbSD is related, *which is true for [almost] every check.*

9.2.2.2. Interpretation of Results of the Qualitative Exploration of the Relationships Calculated by *SecLan*

In the interviews, we made for main observations regarding the relationships calculated by *SecLan*.

Firstly, while inspecting hundreds of relationships, the experts only identified six actual false positives all regarding SonarQube [313]. We extracted the CWEs assigned to the security checks of SonarQube based on the web-sources provided. One example is the check with the ID *RSPEC-6384: Components should not be vulnerable to intent redirection* which is assigned the *CWE-20: Improper Input Validation* [314]. This is especially notable as the expert in I_3 stated that *these false relationships are most likely arising because the checks are pointing to general weaknesses such as CWE-20: Improper Input Validation.* When investigating *CWE-20* [329] in the CWE database, it is especially stated that mapping to this weakness is *discouraged.* Even more it is recommended in this entry to use more specific weaknesses which are part of the root-cause of improper input validation such as *CWE-790: Improper Filtering of Special Elements.* This information shows that the granularity of the CWEs assigned to security checks has a significant impact on the quality of the relationships calculated by *SecLan.* From this information, we conclude that *SecLan* can recommend security checks relevant for detecting security issues in the implementation that could affect the security intended to be achieved by security aspects in the security DSLs.

Secondly, the interview participants often *initially* flagged relationships as false positives which later turned out to be a valid relationship after investigating the reasoning provided by *SecLan.* The expert followed the paths provided by *SecLan* as reasoning for the relationship and investigated the descriptions of the CWEs, the *check descriptions* and the connections between them. The expert in I_2 explicitly stated that *first looking at the [check] description, it was hard to understand, but looking at the explanation [the relation], it's clear.* Similarly, the expert in I_3 stated that the relation is not always immediately clear, *but after looking it [the relation and descriptions of the weaknesses] up, it gets clear why it is related.* This feedback lets us conclude that *SecLan* not only provides relevant relationships between security aspects and security checks but also helps to understand them.

Thirdly, the interviews indicate that even security experts can easily be overwhelmed by the complexity involved in bridging the gap between security DSLs and security checks. This is shown by the fact that in every interview performed, experts flagged some relationships as false positives which later turned out to be valid relationships. They flagged these relationships because they were not aware that a vulnerability detected by a security check could affect the security aspect specified in the security DSL. This challenge also affects the selection of analyses in our analysis coupling. If the coupling expert is not aware of the relationships between the security-related information described in the security DSLs (used for describing the input model of, amongst others, the architectural analysis), and the security checks of the source code analyzers, they cannot properly select the analyses to couple. Thus, *SecLan* can help to overcome this challenge by calculating these relationships and providing reasoning for them.

Despite all these benefits, experts often raised concerns regarding the amount of relationships calculated by *SecLan.* Four experts explicitly discussed the topic of reduction of relationships. The expert in I_1 suggested research into prioritization, e.g., by *considering the level of propagation, e.g., level zero (no step), level one (one step) ....* The expert in I_5 provided similar feedback by stating that *less recommended checks would be ok.* In I_3, the

expert provided a possible reason for some relationships: they could be caused by security checks that are assigned to too general CWEs and that are aimed at broad threats such as *Information Disclosure*. We agree to all feedback about the amount of relationships and discuss future work in Section 11.3 to investigate strategies that reduce them.

We cannot discuss whether we achieved **G4** because we do not state verifiable questions for this goal. Thus, we summarize that our insights based on the interviews suggest that the relationships calculated by *SecLan* are mostly relevant. Furthermore, the provided reasoning helps to understand the relationship between security aspects and security checks. However, the amount of relationships calculated by *SecLan* is an issue that needs to be addressed in future work. We want to highlight one finding that is relevant regarding our analysis coupling. That is, even security experts are not aware of all relationships between security-related information specified in security DSLs and security checks. Consequently, this especially affects the selection of analyses for our analysis coupling because the coupling expert has to expend significant effort to identify and understand these relationships. However, *SecLan* can be used as a building block for overcoming this challenge by calculating the relationships and providing reasoning for them as shown in the interviews. This allows the coupling expert to perform a systematic selection of analyses to couple based on the security aspects specified in the security DSLs.

9.3. Limitations

The evaluation of *SecLan* is subject to several limitations that need to be considered. Some of these limitations also lead to threats to the validity of the evaluation results, which are discussed in Section 9.4. Future work to address these limitations is discussed in Section 11.3. We continue the enumeration of the limitations **L1** - **L21** from the previous sections. In this section, we discuss three additional limitations **L22** - **L24**.

Limitation 22: Limited Number of Participants The number of participants in the survey and interviews is limited. We contacted authors and maintainers of the 66 security DSLs and 33 source code analyzers we studied by email. 132 emails have been sent to the authors and maintainers, of which 17 experts responded for the first part of the survey and 14 responded for the second part. Only seven experts agreed to participate in the interviews. However, these responses are from experts for security DSLs and source code analyzers covering a wide range of security DSLs and analyzers (see Table 9.1).

Limitation 23: Unverified *SecLan* Models We validated the correctness and completeness of the *SecLan* conceptual model instances with experts for security DSLs and source code analyzers. However, from *Limitation 22*, the challenge arises that we could not verify all *SecLan* conceptual model instances created for the 66 security DSLs and 33 analyzers studied due to the limited number of responses from contacted experts. However,

without these validations, the correctness and completeness of the *SecLan* conceptual model instances cannot be guaranteed.

Limitation 24: Limited Number of Explored Relationships Although we explored hundreds of relationships calculated by *SecLan* in the interviews, this is only a small fraction of all relationships calculated (38,136). Still, the explored relationships cover a wide range of security DSLs and source code analyzers. Furthermore, the experts provided feedback about the correctness of the relationships and additional insights about the capabilities and limitations of *SecLan*. However, exploring all relationships is infeasible within the scope of the evaluation and was not possible due to the limited number of participants (see **L22**).

9.4. Threats to Validity

The evaluation of **C3** is subject to several threats to validity. Similar to the discussion of the threats to validity in Section 8.8, we organize our discussion by the threat categories *Internal Validity*, *External Validity*, *Construct Validity*, and *Reliability* of Runeson and Höst [282]. We conducted surveys and interviews with experts to evaluate **C3**, where our findings are often influenced by similar threats to validity.

9.4.1. Internal Validity

Internal Validity ensures that the factors we investigate in the evaluation remain uninfluenced by factors which the researchers are unaware of. The factors we investigated in our evaluation are the correctness and completeness of the *SecLan* conceptual model and whether we can model security DSLs and analyzers with the *SecLan* conceptual model. Furthermore, we explored the relationships calculated by *SecLan*. In this exploration we investigated whether the relationships calculated by *SecLan* are correct and gathered additional qualitative insights about these relationships.

The *SecLan* conceptual model itself could be affected by selection bias due to preferences of knowledge of the authors for particular security DSLs or analyzers. This threat is especially relevant, as we complemented the security DSLs and analyzers identified in surveys with additional ones known to the collaborators. This could have led to a bias in the available participants and on the feedback the participants provide. The same bias has a direct effect on the experts we contacted for the surveys and the interviews. We contacted authors and maintainers of all security DSLs and 33 source code analyzers we studied by email. Thus, to mitigate both threats, we (1) selected the majority of security DSLs and analyzers from surveys, and (2) asked the experts in the interviews for further relevant security DSLs and analyzers. Thus, we see this threat as limited because (1) the majority of the security DSLs and analyzers we studied originate from surveys, and (2)

also the experts in the interviews could not name additional relevant security DSLs or analyzers; indicating that we covered many relevant ones.

An additional threat to validity is the structure of the surveys and interviews which could have influenced the answers of the participants. For the survey, we see this threat as limited, as we primarily used yes-or-no questions and free-text answers to avoid influencing the answers of the participants. Furthermore, we used the same phrasing for investigating all parts of the *SecLan* conceptual model to avoid differences in the understanding of the questions. Additionally, we verified the answers of the participants from the surveys in the interviews. Thus, we see this threat as limited for the survey. Additionally, we provided the same introduction to the interviewee to ensure that every participant has the same understanding of the *SecLan* conceptual model and the topic. Finally, we asked open questions to avoid influencing the answers of the participants.

We addressed the threat of differences in the interviews if they are conducted by different interviewers. We mitigated this threat by having a single collaborator conduct all interviews following the same interview structure. However, from this approach the threat of observer bias remains. We address this threat by always including at least one other collaborator in the interviews and discussing the interview with all collaborators. Therefore, the observer bias should be limited.

We also address maturation effects (i.e., changes in participants' knowledge or attitudes over time) as a threat to validity by conducting all interviews within time frame of three months after the survey. To address this threat, we designed the interview to assess whether the expert had changed insights about the correctness and completeness of the *SecLan* conceptual model compared to their answers in the survey. However, none of the experts changed their insights. Additionally, most experts already had several years of experience in the domain of security DSLs or analyzers, which makes a sudden change unlikely. Consequently, we see this threat as limited.

In our qualitative exploration, we only counted a relationship as correct if the interviewee confirmed its correctness without us interfering. Furthermore, we showed all participants the details of the relationships calculated by *SecLan* and also allowed further questions. Thus, we see the threat to validity as limited here as we could not influence this decision. Also, the result of an expert could be influenced by us so that they would decide that a relationship is correct after investigating the reasoning provided. We see this threat as limited because this behavior has occurred in every interview we conducted. Even more, many interviewees explicitly state that the reasoning provided by *SecLan* helped them to understand why a relationship exists.

9.4.2. External Validity

External Validity concerns the generalizability of our findings.

The generalizability of the correctness and completeness of the *SecLan* conceptual model could be limited by the security DSLs and analyzers we reviewed as this also affects the

selection of experts. Missed security analyzers or checks could have led to missing classes in the SecLan model, inappropriate definitions, or commonalities. In consequence, also an expert in this field could have provided different insights that we missed in our survey and interviews. To avoid this, we primarily based our selection of security DSLs and analyzers on surveys that summarize the state of the art of the research area.

The feedback of the security experts who volunteered to participate in the survey and interview may not be representative, as they are primarily DSL experts. However, although both security DSLs and security checks are widely used in practice [50, 155, 304, 344], commercial tools such as SonarQube have made static security analyzers more widely available and they are now part of many CI pipelines. For this reason, we expect that security experts in general will be more familiar with static security analyzers than with security DSLs. Therefore, it is essential that participants have expertise in security DSLs, which is the case in our study. Furthermore, the experts cover different modeling languages, levels of abstraction, and security aspects, which mitigates threats on the generalizability and applicability of the concepts extracted in our study.

The validity of the explorative interviews could be threatened by factors, such as subject expectancy bias and selection bias. Especially, subject expectancy bias could have affected the interviews, as the *SecLan* was proposed as a solution to bridge the gap between security DSLs and checks. This could have led to the insights only being applicable to the concrete context of the interviews. We mitigated these threats by inquiring about the applicability of the concepts and including multiple interviewers.

One great threat to external validity is the limited number of participants in the survey and interviews. We contacted authors and maintainers of all 66 security DSLs and 33 source code analyzers we studied by email. From these contacted email addresses, 17 experts responded for the first part of the survey and 14 responded for the second part. Only seven experts agreed to participate in the interviews. However, for the security DSLs the experts covered a wide range of abstraction levels and security aspects. For example, we cover business process modeling languages, such as SecBPMN [288], early design data flow modeling languages, such as SecDFD [338], and architecture-level modeling languages, such as UMLsec [154]. Furthermore, the experts cover different security aspects, such as access control, data flow security, and attack modeling. Thus, for the security DSLs we see the threat to external validity as limited. However, for the analyzers the experts cover only a limited set of analyzers. As we invited many experts but only a small number participated, we have to accept this threat. However, we discuss future work in Section 11.3 to increase the number of respondents to address this threat.

9.4.3. Construct Validity

Construct Validity ensures that metrics are capable of answering the evaluation questions. In the evaluation of **C3**, the metrics we applied are mainly the inter-rater reliability *Krippendorff's* α [175], *Percentage Agreement PA* [175, 193], and *Gwet's AC*1 [110] for the correctness of the *SecLan* conceptual model. With these metrics, we test the agreement

between experts regarding the concepts and relations between the concepts in the *SecLan* conceptual model, as well as the correctness of the *Element Types* and relations between them respectively. We see the threat to construct validity as limited for the percentage agreement and Krippendorff's α, as both metrics are widely used to measure inter-rater agreement [175, 193]. Also, different studies [146, 236, 356] investigate Gwet's $AC1$ and highlight its applicability in inter-rater reliability. We specifically counteract the known threat of Krippendorfs's α of prevalence and bias problems [110] by also calculating Gwet's $AC1$. We measure the completeness of the *SecLan* conceptual model by investigating the missing classifications in instances the experts could not remedy with our *SecLan* conceptual model. Collecting missing classifications is an approach used by related work (e.g., [15]) to measure the completeness of conceptual models.

For our qualitative exploration of the relationships calculated by *SecLan*, we did not define specific metrics. We explicitly do only count the false positive results defined by the experts to avoid errors in counting a relationship as correct if the expert is unsure about its correctness. Thus, we see the threat to construct validity as limited here. All other insights are qualitative and cannot be measured with metrics.

The main aspect that could be attributed to threaten the construct validity is the application of surveys and interviews to evaluate the correctness and completeness of the *SecLan* conceptual model and the relationships calculated by *SecLan*. However, we discuss the rationale for applying these data collection methods in Section 7.2. There, we also describe that evaluating conceptual models with expert feedback is seen as reasonable in literature (e.g., by Ananieva et al. [15]).

9.4.4. Reliability

Reliability ensures that results obtained through the data collection and data analysis are independent of particular researchers. Thus, all researchers should also arrive at the same results applying the collection and analysis described in a study.

We documented for which security DSLs and analyzers we reached experts for the survey and the interviews conducted. However, we had to ensure the anonymity of the participants. Thus, we cannot provide a complete list of all experts we contacted. Nevertheless, we provide a thorough documentation of the security DSLs and analyzers we contacted experts for, allowing to reproduce the contacted experts in the surveys and interviews. Reliability can be threatened by differences in the survey and interview questions. This is why we provide other researchers with the surveys and interview guidelines we applied in our evaluation. Thus, other researchers can reproduce the surveys and interviews we conducted. Similarly, reliability can be threatened by researchers using different instances of the *SecLan* conceptual model and the resulting calculated relationships, even when asking the same experts. Thus, we provide other researchers with the created instances of the *SecLan* conceptual model for the 66 security DSLs and 33 source code analyzers we studied in our replication package [270]. In addition, we provide the *SecLan* tool so that other researchers can reproduce the relationships used based on these instances. We

calculate the inter-rater reliability metrics using the statistics tool *R* [59]. We provide other researchers with the survey results in form of matrices and an *R* script we applied to calculate the metrics in our replication package [270]. This leaves hardly any threat to reliability in calculating the inter-rater reliability metrics with *R* based on the results we obtained in the survey. Unfortunately, we cannot provide the raw data of the surveys and interviews due to the anonymity of the participants. However, we provide excerpts of the interviews in Subsection 9.2.2.

9.5. Summary and Outlook

In this chapter, we presented the evaluation of *SecLan* (**C3**). The evaluation goals were to (1) assess the *correctness and completeness* of the *SecLan* conceptual model and its instances (**G3**), and (2) qualitatively explore the relationships calculated by *SecLan* (**G4**).

We begin by presenting the study design. In this design, we use surveys and expert interviews to evaluate the *SecLan* conceptual model and the relationships calculated by *SecLan*. The study design includes the participant selection, the design of the survey and interviews, and details about how we operationalize the calculation of the metrics **M3.1.1**, **M3.1.2**, and **M3.1.3** for evaluating the correctness of the *SecLan* conceptual model (**Q3.1**, **Q3.2**). Based on the study design, we then present the survey results and their interpretation regarding the correctness and completeness of the *SecLan* conceptual model and its instances. Furthermore, we present the results and their interpretation of the qualitative exploration of the relationships calculated by *SecLan* using the interviews. Finally, we discuss the limitations of our study design and the resulting threats to validity.

Participant Selection We contacted authors and maintainers of the 66 security DSLs and 33 source code analyzers we investigated in the synthesis of the *SecLan* conceptual model (see Subsection 6.1.1) by email. Of all contacted experts, 17 participated in our survey and 7 participated in interviews.

Survey Design We use expert surveys to evaluate the correctness and completeness of the *SecLan* conceptual model and its instances (**G3**). We design two surveys; the first survey targets experts in security DSLs and the second survey targets experts in analyzers.

The surveys are structured similarly and comprise four parts: an introduction, two core parts, and a follow-up request for participation in our interview. The first core part assesses the experts' opinion regarding the correctness of the *SecLan* conceptual model. This part contains a total of 12 questions, where some questions are only asked if the expert provides a certain answer. For example, if an expert's answer about the correctness of concepts implies disagreement, we ask which concepts are not correct. Thus, up to six questions assess the correctness of the concepts and relations between the concepts in the *Security Model*, the *Security DSL Description* and the *Security Analyzer Description*. Up to two questions assess the correctness of the *Element Types* and relations between these *Element*

Types we provide in our *System Model*. Up to four questions assess the correctness of the relations from the *Security DSL Description* and *Security Analyzer Description* to the *Security Model* and the *System Model*.

The second core part assesses the completeness of the *SecLan* conceptual model and the correctness and completeness of instances of the *SecLan* conceptual model created by us. We ask the experts 15 questions regarding security DSLs and 19 questions regarding analyzers. Similarly to the correctness of the *SecLan* conceptual model, if an answer of an expert implies that the instance is incorrect or incomplete, we ask further questions to obtain further details. Every survey contains these questions regarding security DSLs and analyzers. However, the survey targeting experts in security DSLs first asks the questions regarding security DSLs and then those regarding analyzers; the survey targeting experts in analyzers asks them in reverse order. With this design, we aim to maintain motivation and counteract fatigue effects. The questions about security DSLs and analyzers can be answered up to three times each to account for experts having expertise in multiple security DSLs or analyzers.

We first ask up to three questions for the security DSL or analyzer the experts want to answer questions about and assess their level of expertise with them. We ask up to two questions to assess whether there are elements in the specific security DSL or analyzer that we did not address. Furthermore, we ask up to two questions to assess whether the expert understands the classification. Then, we ask up to eight questions to assess the correctness and completeness of the instances we created. For the analyzers, we also ask up to four questions to assess the correctness and completeness of the specific weaknesses we assigned to their security checks.

In the final part of the survey, we ask the experts whether they would like to participate in the qualitative exploration of the relationships calculated for a security DSL or analyzer of their choice. Thus, we ask up to two questions to determine whether they want to participate and to collect contact details.

Interview Design The main purpose of the interviews is to qualitatively explore the relationships calculated by *SecLan*. Each interview is time-boxed to one hour, meaning that we continue the exploration only for a fixed time. The interview structure contains five parts. First, we provide an introduction to the interview, again explaining the context of *SecLan* and asking about open questions. Second, we assess whether the feedback provided by the expert in the first core part of the survey still reflects the expert's understanding of the *SecLan* conceptual model. Third, we assess whether the instance of the security DSL or analyzer for which we aim to explore relationships is correct. Here, we aim to address challenges arising from additional insights gained by the expert or challenges arising from changes we made to the instance based on the expert feedback in the survey. Fourth, we iteratively explore the relationships calculated by *SecLan* for the security DSL or analyzer selected by the expert. We start by selecting relationships to security DSLs or analyzers the expert is aware of. Then, we continue exploring relationships iteratively

until ten minutes before the end of the one-hour session. Finally, we ask the experts about remaining questions and remarks, and thank them for their participation.

Calculation of Inter-Rater-Agreement For evaluating the correctness of the *SecLan* conceptual model, we apply inter-rater agreement metrics. To calculate these metrics, we encode expert responses obtained in the survey regarding the correctness of concepts, *Element Types*, and relations into matrices. For sub-models for which an expert did not provide answers, we remove the corresponding row. For every sub-model of the *SecLan* conceptual model, we provide a dedicated matrix. We then apply library functions in the statistics tool *R* to calculate Krippendorff's α, Percentage Agreement (PA), and Gwet's $AC1$ for each matrix.

Results and Interpretation: Correctness and Completeness of the *SecLan* conceptual model 17 experts answered the survey regarding the correctness and completeness of the *SecLan* conceptual model and its instances. All experts provided feedback regarding the correctness of the *SecLan* conceptual model, and 15 experts provided feedback regarding the correctness and completeness of the instances we created for the security DSLs and analyzers.

16 experts fully agreed with the concepts *security objectives, threats, weaknesses* and their relations in the *System Model*. One expert expressed concerns about the relation between threats and weaknesses. For the *System Model*, 15 of 17 experts fully agreed with the *Element Types* and relations; two experts raised minor concerns about specific relations that they confirmed as valid in interviews. 15 of 17 experts provided feedback for the *Security DSL Description*. All 15 responding experts agreed on concepts *Security DSL*, *Security Specification* and the relationships between the *Security DSL Description* and the *System Model*. However, three experts expressed concerns about the terminology of *Security Aspect* (suggesting "Security Control" or "Countermeasure" instead) and its relation to the *Security Model*. 16 of 17 experts provided feedback for the *Security Analyzer Description*. From these experts, 15 fully agreed with the concepts, relations between these concepts, and the relations to the *System Model* and *Security Model*. One expert expressed concerns regarding the relation between *Security Check* and *Element Type*. Here, due to the statement provided, we assume a mismatch in terminology. Overall, high inter-rater agreement was achieved with PA values between 0.800 and 0.939, and Gwet's $AC1$ values between 0.954 and 0.992, with 7 disagreements out of 863 possible answers. Krippendorff's α showed low values (between -0.011 and 0.121) due to the highly homogeneous agreement between raters, which is penalized by Krippendorff's α.

14 experts provided feedback regarding the correctness and completeness of the instances we created for the security DSLs and analyzers. The responses cover a wide range of security DSLs (CARDS [150], Attack-Fault Trees [180], PbSD [2], SecBPMN [288], SecDFD [338], SecureUML [192], SysML-Sec [17], UMLsec [154] (in three responses), the Architectural Data Flow DSL [300] and the Attack Propagation DSL [348]). We received responses for two analyzers (FlowDroid [19], and PMD [260]). Four experts provided feedback about classifications that were missing. Two experts mentioned concepts in their security DSLs

that could not be expressed with the *SecLan* conceptual model (countermeasures and indirectly derived mitigations). One expert mentioned that FlowDroid provides a building block for analyses and, thus, cannot be fully described with the *SecLan* conceptual model.

From this feedback, we derived two core challenges: (1) the need to express countermeasures and indirectly derived mitigations in the *SecLan* conceptual model, and (2) that we cannot properly express security DSLs that already annotate vulnerabilities or weaknesses to the system to perform analyses (e.g., the Attack Propagation DSL [348]).

Still, due to the high agreement regarding the concepts and relations in the *SecLan* conceptual model, we conclude that the *SecLan* conceptual model is correct. Furthermore, only two experts mentioned concepts in their security DSLs that could not be expressed with the *SecLan* conceptual model, and most other missing classifications could be expressed with existing concepts or disregarded because of a terminology mismatch between the experts and the *SecLan* conceptual model. Thus, we conclude that the *SecLan* conceptual model and its instances are mostly correct and complete.

Results and Interpretation: Qualitative Exploration of Relationships Seven security DSL experts participated in exploring hundreds of calculated relationships between security aspects and analyzer security checks. Only six false positives were identified, all from SonarQube. One expert stated that the false positives probably arise due to mapping security checks to overly general CWE assignments. Notably, in every interview, experts initially flagged relationships as false positives but later acknowledged them as valid after investigating the reasoning (paths and CWE descriptions) provided by *SecLan*. Experts in five interviews explicitly stated that the provided paths helped clarify relationships. Four experts raised concerns about the volume of relationships, suggesting prioritization or filtering strategies. These findings demonstrate that *SecLan* successfully identifies mostly correct relationships, helps experts understand the design-implementation gap, and reveals that even security experts lack awareness of all relationships between DSLs and analyzers.

Limitations Three limitations are identified. First, we could only acquire a limited number of survey participants (17 for correctness, 14 for instances, 7 for interviews) from 132 we contacted. This number limits generalizability of the results obtained in the study. Still, the participants cover a wide range of DSLs and analyzers. Secondly, we could not verify every *SecLan* conceptual model instance we created due to the limited participant responses. Consequently, we cannot claim correctness and completeness for them. However, the verified instances cover a representative sample of the studied DSLs and analyzers. Thirdly, we could only validate a fraction of all relationships calculated by *SecLan*, due to the limited number of interview participants and the time-bounded interview duration.

Threats to Validity We discuss threats to validity of our evaluation, structured by the threat categories *Internal Validity*, *External Validity*, *Construct Validity*, and *Reliability* of Runeson and Höst [282].

Internal Validity concerns factors that may unknowingly influence our findings. In this category, the selection of security DSLs and analyzers can introduce selection bias, which we mitigated by relying primarily on state-of-the-art surveys and by asking experts for missing candidates. Another threat is the structure and phrasing of survey and interview questions, which may influence participant responses. We mitigated this threat by using consistent phrasing and allowing free-text answers. Wherever possible, we also validated survey responses during interviews. The interviews themselves can introduce bias due to differences in interviewers and questioning. We mitigated this threat by using a fixed interview structure, assigning one main interviewer, and including at least one additional collaborator in each interview. Furthermore, all responses and their interpretations were discussed among the collaborators involved. Finally, maturation effects between the survey and interview phases can influence responses. We mitigated this threat by limiting the interview period to three months after the surveys.

External Validity concerns the generalizability of our findings. In this category, threats arise from potentially missing analyzers or checks in the literature, the limited number of participating experts, and because the experts are primarily DSL experts. We mitigated these threats by selecting security DSLs and analyzers based on state-of-the-art surveys. Furthermore, the reviewed security DSLs cover various abstraction levels and security aspects. We also assume that experts in security DSLs are familiar with analyzers, as analyzers are widely used in practice and integrated into many CI pipelines. Still, the limited number of participating analyzer experts remains a relevant threat to generalizability.

Construct Validity concerns whether our evaluation methods appropriately answer our research questions. Threats include the suitability of inter-rater agreement metrics, the measurement of completeness via missing classifications, the qualitative exploration of relationships without quantitative quality metrics, and the general suitability of surveys and interviews for our research questions. We mitigate the threat affecting inter-rater agreement by using established metrics (Krippendorff's α, PA, and Gwet's $AC1$). We explicitly address known assumptions of Krippendorff's α regarding response distributions by also reporting Gwet's $AC1$. To address the threat regarding the measurement of completeness, we explicitly ask experts about missing classifications and report them in the results. We also note that related work already applies surveys and interviews to evaluate the correctness and completeness of conceptual models.

Reliability concerns reproducibility and independence from individual researchers. Threats to reliability include the possibility that other researchers contact different experts. Although we cannot provide a complete list of all experts we contacted to ensure anonymity of the participants, we provide a list of all security DSLs and analyzers we contacted experts for. Another threat is asking different questions in surveys and interviews, which may lead to different responses. We mitigate this threat by providing the exact survey and interview questions in the replication package [270]. Finally, other researchers may create different instances of security DSLs and analyzers, which may lead to different results. We mitigate this threat by publishing the created *SecLan* conceptual model instances in the replication package [270].

Outlook This chapter concludes *Part III - Validation* of this thesis. We next proceed with *Part IV - Epilog* where we first discuss work related to the contributions presented in this thesis in Chapter 10. Finally, we conclude this thesis in Chapter 11 with a summary of the thesis, a discussion of the benefits of our contributions, and a presentation of various possible future work.

Part IV.

Epilog

10. Related Work

Literature: This chapter is based on the following (co-) authored publications: [IEEE TSE 2025], [IEEE ICSA-C 2025], and "*Can I Check What I Designed? Mapping Security Design DSLs to Code Analyzers*" [254] (major revision under review in the journal *ACM Transactions on Software Engineering and Methodology* at the date of submission of the dissertation)

In this chapter, we discuss work related to our contributions presented in this thesis; namely the coupling design (**C1**), the coupling approaches (**C2**), and *SecLan* (**C3**). As the coupling design and coupling approaches are strongly related, we discuss related work for both contributions jointly. Although *SecLan* is a separate contribution, most related work is common with the coupling design and coupling approaches.

One topic common to all contributions is compliance between design models and the implementation, for which we discuss related work in Section 10.1. This topic is relevant for the coupling design and coupling approaches, as they rely on detecting non-compliance between security characteristics specified in the *Annotated Architecture* and the implementation to calculate the security characteristic resulting from the implementation. This topic is also relevant for *SecLan*, as the relationships between security aspects and security checks indicate whether a check can detect security vulnerabilities that result in non-compliance with the security properties specified in security DSLs and the implementation.

Another topic common to all contributions is analysis exchange formats, which are reflected in the reference metamodels defined in the coupling approaches and the *SecLan* conceptual model defined in *SecLan*. We discuss related work regarding analysis exchange formats in Section 10.2. These formats are important to enable systematic representation of analysis results so that they can be used by various analyzers or to relate the results to other information (e.g., relate the information to system elements).

Specific to the coupling design and coupling approaches is related work on relating analysis results to system other system abstractions, discussed in Section 10.3. Although this topic is a subset of compliance — or non-compliance detection — between design models and implementation, it is relevant enough to be discussed separately because it also includes related work that does not focus on compliance. We specifically discuss related work that focuses on lifting analysis results to system representations different from those used by the analysis tool, as this is a key aspect of our coupling design and coupling approaches.

Specifically related to *SecLan* are approaches that capture security knowledge. This work is related because we use relationships between security aspects and security checks to

capture knowledge about the relation between security properties specified in security DSLs and the results of source code analyses. We discuss related work in this domain in Section 10.4.

Naturally, approaches for analysis coupling are related to our coupling design and coupling approaches. Consequently, in Section 10.5, we discuss related work regarding analysis coupling, independent of the specific analyses being coupled.

For all related work discussed, we summarize its relation to our contributions at the end of each subsection.

We omit related work in the domain of model synchronization, such as consistency preservation [167, 245], or bidirectional languages and model transformations [129, 130, 297], as these approaches do not focus on analysis results and their relation to other artifacts, but rather on synchronizing models. We also omit related work in the domain of isolated security analysis which we see as foundation for the analysis coupling rather than as related work.

10.1. Compliance between Design Models and Code

The topic of compliance is relevant, as our coupling design and coupling approaches rely on calculating the security characteristic resulting from the implementation based on source code analysis results that indicate non-compliance (*Analyses Input Model Consistency* and *Non-Compliance Analysis* step; see Section 4.4). This topic is also relevant to *SecLan*, as it relates security DSLs with source code analyses that are capable of detecting non-compliance between security properties specified in security DSLs.

The approaches in this domain most closely related to our contributions analyze non-compliance between design artifacts and the implementation in the domain of security using source code analyses. However, these approaches all differ from our contributions, as they use specific design models and specific source code analyses to detect non-compliance.

Work related to our specification-based non-compliance detection includes CARDS [150], IFlow [159], as well as the approaches by Töberg et al. [334], Muntean et al. [222], and Yurchenko et al. [363]. These approaches analyze non-compliance by deriving specifications for information flow analyses and by mapping design models to source code. For example, CARDS derives specifications for information flow analyses from design models and then uses these specifications to check whether the implementation complies with the design models. However, these approaches are tailored to specific design models and specific analyses, while our coupling design and coupling approaches are formulated independently of specific analyses, their models, and security characteristics.

Several related approaches [250, 256, 337] analyze non-compliance between security data flow diagrams (SecDFDs) and the implementation. Among other aspects, they detect whether encryption is present or absent in the implementation as described in the SecDFDs.

This work is related to ours, as it detects non-compliance regarding encryption. However, these approaches focus on *design-related non-compliance* (i.e., non-compliance between elements described in the design models and those implemented). In contrast, our analysis coupling focus on the detection and utilization of *specification-related non-compliance* (i.e., non-compliance where the implementation is realized so that it does not comply with the specified security characteristics). Schneider et al. [296] apply *specification-related non-compliance* detection by analyzing whether information flows in the implementation comply with the endpoints defined in the Node-RED specification. This application is related because it analyzes *specification-related non-compliance* between specifications and implementation with the source code analysis *CodeQL* [100, 214]. However, this approach is tailored to Node-RED specifications and a single source code analysis (i.e., CodeQL). Furthermore, this approach uses queries specific to Node-RED nodes, and it does not use the specification of security characteristics to detect non-compliance.

Other approaches do not analyze non-compliance between design models and the implementation by deriving specifications for source code analyses, but rather follow the secure-by-construction approach where design models are used to construct code that complies with the design models by construction. We see the work of Runge et al. [283] as notable in this domain, because it provides an approach to ensure information flow security by construction.

Jasser [145] analyze non-compliance between architectural security rules specified in a language and the implementation using dynamic analysis. In contrast, we focus on static analysis to detect non-compliance between security characteristics specified in design-time security DSLs and the implementation, which allows avoiding potential security risks arising from the deployment necessary for dynamic analysis. Abi-Antoun et al. [1] calculate compliance between a high-level DFD, provided by the user and annotated with security properties, and the implementation. In this approach, first a DFD is extracted from the implementation and then checked for compliance with the high-level DFD. UMLSecRT [251] is an approach for detecting non-compliance between UMLSec models and the implementation at runtime. In this approach, the UMLSec models are used to generate runtime monitors by bytecode instrumentation. These monitors check whether the accessed member, the accessing member, or both are missing annotations, resulting in a violation of the *secure dependency* security characteristic specified with UMLSec stereotypes. This approach is related to our coupling, as it detects non-compliance between design models and the implementation, but differs by focusing on runtime analysis and by being tailored to specific analyses (the checks implemented in the monitors) and specific security characteristics (the secure dependency stereotype of UMLSec).

There are also several approaches that analyze non-compliance between design models and the implementation by generating test cases from design models and executing these test cases on the implementation. Xu et al. [359, 360] generate executable security tests from threat models specific to their approach. From these models, they generate the test initialization code, the test input code, and the test oracle code, e.g., for HTML/Selenium, C, or Java. Marback et al. [199] generate valid and invalid inputs from threat trees and

further design documentation. These inputs are used by generated test scripts that execute test sequences on the system under test.

In the domain of security, Ferrara et al. [90] use taint analysis to detect non-compliance between General Data Protection Regulation (GDPR)-related privacy policies and the implementation. Similarly, M. Binder and Kunz [194] extract code properties for different parts of the GDPR, e.g., data flows and database operations. These properties are encoded into a privacy property graph that is used to perform a (semi-)automated compliance analysis. Kellogg et al. [160] present an approach for continuously checking compliance with certain security properties during development. This approach implements various security analyses to check compliance regarding specific *compliance regimes* (i.e., security standards). This work is related to our contributions, as it mainly focuses on compliance with encryption-related properties such as cryptographic key length, usage of HTTP instead of HTTPS, and usage of hard-coded secrets.

Less closely related are approaches that analyze compliance between design models and implementation without a focus on security. Consequently, these approaches are naturally distinct due to the lack of a security focus. The general topic of architectural erosion — i.e., the non-compliance between architectural design and implementation — is related to our contributions, as it also focuses on compliance between design models and implementation. In this domain, Li et al. [189] present a systematic mapping study on architectural erosion, where many approaches focus on detecting divergences between architectural design and implementation without a specific focus on security. For example, Zhong et al. [368] investigate compliance between a domain model for microservice architectures and the implementation. In this approach, Zhong et al. define patterns for the domain model and the implementation, and check whether the patterns are present in both artifacts to detect non-compliance. Similarly, Sun et al. [321] analyze compliance between the expected behavior of microservice-based systems and the actual behavior. For this purpose, they provide a dedicated design model for microservice-based systems. This model is instantiated with the expected behavior of the system and then used to generate test cases that are executed on the implementation to check whether the expected behavior is compliant with the actual behavior. For this, expected execution traces and actual execution traces are compared to detect non-compliance. Both approaches are distinct from ours by focusing specifically on the domain of microservices and by using tailored analyses to support non-compliance detection.

Uzun and Tekinerdogan [339] model criteria for architecture compliance analysis and generate test cases from these criteria. For this purpose, specific mapping rules are defined that have to hold to allow non-compliance detection. Díaz-Pace et al. [66] use heuristics to check compliance between architectural scenarios and the implementation. The architectural scenarios are specified in use-case maps that comprise structural and behavioral aspects of the system. Based on this, a reviewer executes test cases to check compliance. Passos et al. [244] compare three techniques for checking compliance between architecture and implementation. In particular, they use *.QL* [214] queries to check whether source code elements in the implementation depend on elements that are not allowed according to the architectural design. Marchezan et al. [200] uses consistency preservation

rules to check compliance between UML [235] models and the implementation. This approach is notable because it abstracts target information in UML models and in the implementation with a common metamodel, which is similar to our reference metamodels. Still, this approach focuses on reporting non-compliance rather than investigating the impact of non-compliance on design-level properties, which is the focus of our coupling design and coupling approaches.

In summary, related work on non-compliance detection often follows a similar pattern by generating information for source code analyses from design models, which then detect violations that imply non-compliance. However, most approaches we found in the domain of security are tailored to specific design models and specific analyses, while our coupling design and coupling approaches are formulated independently of specific analyses, their models, and security characteristics. Thus, our coupling approach is a step further to generalize the detection of non-compliance between design models and implementation, which is a prerequisite for analysis coupling. Other approaches focus on run-time analysis, while we focus on static analysis to avoid potential security risks arising from the deployment necessary for dynamic analysis. Our contributions are independent of specific analyses and security characteristics, which better supports coupling other pairs of analyses without requiring coupling experts to understand a specific analysis coupling. SecLan *is also independent of specific analyses and security characteristics, as it provides a systematic approach to relate security properties specified in design-time security DSLs with source code analyses.*

10.2. Analysis Exchange Formats

There are several formats for the systematic representation of analysis results and their exchange between different tools.

The OASIS SARIF [16] provides a standardized format that allows integration of static analysis tool results in different environments, such as consolidated representations in an IDE. The ASSUME Static Code Analysis Tool Exchange Format [161] is another format that enables comparability between analysis results. Highly related to our reference metamodel for the *Source Code Analysis Result* in the specification-based coupling approach is the RIFL [82]. This language allows the standardized representation of information flow analysis results. For this purpose, RIFL provides constructs such as sources, sinks, and domains; domains are analogous to our security characteristics. In contrast, our reference metamodel is not limited to sources and sinks and also captures the applied configuration.

Haltermann and Wehrheim [114] present an exchange format for source code analysis results based on a Generalized Information Exchange Automaton (GIA). This automaton is represented by a graph in which nodes are states in the implementation and edges define transitions between them. This graph can be used by source code analyses to show paths in the implementation that lead to a target state, paths that do not lead to a target state, or paths for which it is unknown whether they lead to a target state but that are

potentially interesting. The GIA is related to our coupling model in the pattern-search-based coupling approach, as both approaches represent the implementation in a graph structure to facilitate communication between analyses. However, our coupling model is specifically designed to capture the information that allows calculating non-compliance between security characteristics specified in architectural models and the implementation. In contrast, GIA is used to generally represent source code analysis results.

Düsing and Hermann [72] found that the lack of standardized exchange formats for static analysis results leads to a high effort for integrating different analyses into a single environment. Therefore, they propose an exchange format for static analysis results which can be used to publish, search and reuse analysis results. This format is related to our reference metamodels— in particular the source code analysis output —, as it abstracts the output of source code analyses into a systematic format. However, our reference metamodels are designed to capture information that allows relating source code analysis results to architectural models and security DSLs, while the format of Düsing and Hermann [72] are designed for more general purposes of representing and exchanging analysis results on the source code level.

In summary, all exchange formats we found target information exchange on the same level of abstraction, while our reference metamodels and the SecLan *conceptual model are designed to relate source code analysis results to the architectural level or security DSLs.*

10.3. Relating Analysis Results to Other System Representations

Another related topic is lifting analysis results to system representations different from the one used by the analysis tool. In the coupling design and coupling approaches, we have to relate and integrate source code analysis results with design models to calculate the security characteristic resulting from the implementation. Here, the source code analysis takes the implementation as input, and the resulting information (i.e., detected vulnerabilities) is related to design models. Durán et al. [69] refer to making analysis results available to other tools as *lifting*.

Many approaches in this domain focus on analyzing properties of a user-facing model with an analysis using another formalism and then lifting the results back to the user-facing model. In one version of the DFA [113, 300], the PCM, security characteristics, and data flow constraints are transformed into Prolog facts. Then, Prolog rules are used to analyze whether the data flow constraints are violated. The results of the analysis are represented as Prolog facts that are transformed back to the PCM to relate the detected violations to model elements of the PCM. Gerking et al. [95] transform models of the domain-specific modeling language mechatronic UML, including timing-related constraints regarding the modeled system, into timed automata for the model-checking tool UPPAAL. UPPAAL provides counter-examples if timing constraints are violated, which are then lifted back and related to model elements of mechatronic UML. Similarly, Buchholtz et al. [41] transform

UML models into an input model for the LySatool to detect protocol violations. If such violations are detected, the results are lifted back to the UML models to relate violations to model elements. Thus, these approaches differ from ours, as they focus on lifting within a single development phase, e.g., architecture or implementation. In contrast, our approaches focus on lifting analysis results from the implementation phase to architectural models created in the architectural phase.

Related work on this topic in the domain of security is presented by Kirschner et al. [165], who use the source code analysis Snyk to detect vulnerable dependencies in the implementation and lift these results to the user-facing architectural model of Walter et al. [348]. In this architectural model, the detected CVE IDs are used to annotate components affected by vulnerable dependencies. As they relate vulnerabilities to the security DSL, this approach is related to our coupling design and coupling approaches, as well as to *SecLan*. However, they again focus on specific analyses and the vulnerability elements in the security DSLs, while we provide a more general approach independent of specific analyses and security characteristics. Lanzinger et al. [182] present an approach to quantify software correctness by combining architectural modeling and formal program analysis. In this approach, Lanzinger et al. integrate information about coverage regions — combinations of parameter and field values for which program correctness could be verified — into an extended version of the PCM that can capture these regions. This approach is distinct from our contributions, as it uses architectural elements specifically designed to capture correctness information, whereas we do not modify the metamodel of the architectural analysis. Peldszus et al. [253] present an approach to prioritize source code analysis results based on requirements. This approach is related to our contributions, as it relates analysis results to design models. However, we see our contributions as complementary to this work due to different goals. First, Peldszus et al. do not consider the impact of non-compliance between design models and implementation on the analysis results. Thus, our coupling could be used as an additional information source to improve the prioritization of analysis results. We discuss this idea in more detail in our future work in Section 11.3. Secondly, *SecLan* can be used to provide security analyses that have an impact on the design models used for prioritization.

When not considering security, the approaches iObserve [120] and CIPM [213] are also related work. These approaches use monitoring information from the implementation to reflect changes in the system by updating an architectural model. In contrast, we focus on integrating information retrieved with security analysis at design time. Performing security analysis at design time is important because an adversary cannot exploit a potential vulnerability in a deployed system while trying to detect it, which is necessary for dynamic analysis. Cortellessa et al. [54] refactor user-facing models based on the results of an analysis performed with a non-functional analysis model. For this purpose, the non-functional analysis model is generated from user-facing UML models and analyzed regarding availability properties. The non-functional analysis model is refactored to fix detected violations, and the changes are lifted back to user-facing UML models using bidirectional transformation. Similarly, Meyers et al. [209] present a framework for verifying temporal properties in a discrete-time behavioral domain-specific modeling language. They define five sub-languages to capture different information necessary for

their approach, comprising, e.g., a language to capture static system structure, a language to capture information that can change at runtime, and a language to specify temporal or structural properties. Based on this language, they define a transformation into an input model for a model checker to verify temporal properties. They illustrate their approach by generating a Promela model, which is analyzed with the model checker Spin. If the model checker detects a violation of the specified properties, a counterexample is generated and lifted back to the original model of the domain-specific modeling language by using model element IDs. Furthermore, Hegedüs et al. [118] present an approach for representing simulation traces in traces of the design model, called back-annotation. This approach is illustrated by analyzing BPEL [233] processes using a Petri net simulator. From the BPEL process, a Petri net is generated. In this step, traceability information is captured because back-annotation is not possible solely with information from the Petri net. Finally, micro and macro traces in the Petri net simulation are generated and back-annotated to the BPEL process using model transformations. Similar to the approach of Buchholtz et al. [41], Shah et al. [302] transform UML models into an input model for the Alloy Analyzer to detect violations of properties specified in the UML models. After performing the analysis, the Alloy models are transformed back to UML models. All approaches are related to our contributions, as they also focus on lifting analysis results to a model in a design language. However, besides the missing focus on security, they differ from our coupling design and coupling approaches by analyzing properties of the user-facing model with an analysis using another formalism and then lifting the results back to the user-facing model, while our analysis coupling focuses on analyzing system abstractions available at different development phases and then relating the results to each other.

Note how many of the round-trip approaches [41, 54, 95, 209, 300, 302] comprise a process similar to our coupling process: they transform an initial model into a model suitable for analysis, perform the analysis, and then lift the results back to the initial model. However, these approaches differ by either focusing on artifacts within a single development phase or by not detecting non-compliance.

Esposito et al. [83] present a study on the correlation between analysis results of static code analyses and architectural smells. They use ARCAN [92] to detect architectural smells and four static code analyses, amongst others, them SonarQube [313] and PMD [260], to detect code smells. Based on the results of these analyses, they analyze the correlation between code smells and architectural smells. This work is related to our contributions, as it relates analysis results from two different system abstractions. However, they neither focus on security nor present defined mappings to relate analysis results, but rather analyze correlations between the results of the analyses.

In summary, most related work uses lifting of results to present them to the user in a user-facing model (e.g., violations calculated in prolog in an architectural model), which is applied in the same development phase as the user-facing model (e.g., architectural models). In contrast, our coupling design and coupling approaches focus on lifting analysis results from the implementation phase to architectural models created in the architectural phase. Other related work which lifts analysis results to system abstractions commonly created in other development phases (e.g., Kirschner et al. [165] and Lanzinger et al. [182]) is tailored to

specific analyses and specific user-facing models, while our contributions abstract from specific analyses and specific models. There is also related work that is not located in the domain of security, which is distinct from our contributions by the lack of a security focus. Consequently, related work differs from our contributions by either focusing on lifting within a single development phase, by being tailored to specific analyses and user-facing models, or by not focusing on security.

10.4. Security Knowledge Representation

There are different approaches to systematically capture security knowledge. Most notable is the approach of Ruiz et al. [281], which captures security knowledge using a core security metamodel. This metamodel captures different aspects, such as security solutions and threats. Most notably, the metamodel can be structured into several sub-metamodels that have similarities to information captured in *SecLan*. For example, the property sub-model captures security properties, amongst others confidentiality, integrity, and availability properties, which are also captured in *SecLan* as security objectives. Furthermore, the threat sub-model captures different types of threats and attacks implementing these threats; this information is captured in *SecLan* as the threats of the STRIDE [304] threat model and the corresponding weaknesses that enable these threats. In particular, the validation and verification sub-model connects analyses to attacks in the threat sub-model, which is similar to the relationships between security checks and weaknesses in *SecLan*. Thus, although the approach of Ruiz et al. [281] is similar to *SecLan* by capturing knowledge about security properties (i.e., security objectives), threats, and attacks (analogous to weaknesses in *SecLan*), it is intended to capture this information as a knowledge base. In particular, this approach contrasts with *SecLan*, as it does not include the notion of security DSLs, and the connection between analyses and how to achieve security (security properties) has to be provided manually by an engineer. In contrast, we use the information in the models to systematically relate security properties specified in security DSLs with the results of source code analyses to detect non-compliance between security properties and the implementation.

Security knowledge is also captured by various security standards, such as NIST SP 800-160 [279] and the ISO/IEC 27000 series [139, 140]. However, these standards are used to capture security knowledge more generally and are not tied to relating security properties to source code analyses, which is the focus of *SecLan*. Similarly, security knowledge is also represented using security ontologies, as shown in the systematic literature review by Adach et al. [3]. For example, Herzog et al. [128] capture the six core concepts of threats, vulnerabilities, security goals, defense strategies, assets, countermeasures, and relations between these concepts. They provide specific classes for these concepts, e.g., 88 threat classes, 79 asset classes, and 133 countermeasure classes. In particular, the concepts of security goals, threats, and vulnerabilities are related to the concepts of security objectives, threats, and weaknesses in *SecLan*. Similarly, Pereira and Santos [258] provide a security ontology with concepts like CIA, threat, vulnerability, or incidents. Consequently, although

several ontologies capture concepts similar to those in *SecLan*, they are tailored to use cases that differ from that of *SecLan*; namely, they do not capture concepts such as security checks or security DSLs.

In summary, while various related-work approaches exist for capturing security knowledge, these approaches differ from SecLan *because they are not tied to relating security properties to source code analyses, which is the focus of* SecLan*. To an extent, these approaches can be used as a knowledge base to populate the models of* SecLan*, but they do not provide the systematic approach for relating security properties specified in design-time security DSLs to source code analyses that* SecLan *provides.*

10.5. Approaches for Analysis Coupling

Naturally, an important topic related to our contributions is the set of approaches that enable the coupling of analyses. Heinrich et al. [122] discuss fundamental properties in model-based analysis composition, which are also used as a foundation of this thesis and are closely related to our analysis coupling (see Section 2.5).

OPAL [125] is an approach for coupling static program analyses by using grey-box coupling (i.e., exchanging information during the execution of the analysis algorithms). Based on OPAL, Roth et al. [280] present a framework for coupling static program analyses across different programming languages. Lerner et al. [187] presents a framework for coupling data flow analyses where the analyses can perform graph transformations by replacing information in a graph structure with a sub-graph. Both approaches focus on coupling source code analyses and using intermediate results provided by analyses at the same level of abstraction. This contrasts with our coupling design, which focuses on coupling architectural analyses and source code analyses.

Johnson et al. [152] presents a framework for collaborative dependence analyses where complex queries are decomposed into simpler queries that are answered by different analyses and then combined to answer the complex query. Similarly, Pauck and Wehrheim [246] present an approach for cooperative Android app analysis where different analyses are used to perform different tasks in an information flow analysis. Again, both approaches focus on coupling source code analyses used for the same system abstraction by decomposing complex queries into simpler queries that are answered by different analyses. This contrasts with our coupling design, where analyses are not decomposed into simpler analyses but rather perform their analyses independently and then exchange information to calculate the security characteristic resulting from the implementation. Also, Richter et al. [277] present an approach for cooperative software verification (a type of static analysis) where different verification tools are used to perform a verification task. If the verification task is deemed hard by one verifier, it splits the program into sub-programs and tries to verify the sub-programs. If the verification cannot be performed by the verifier, the unfinished verification tasks are passed to another verifier. This procedure is repeated until the verification task is completed or all verifiers have tried to verify the program.

Dwyer and Elbaum [73], and Cruanes et al. [56] define approaches that focus on making the results of different analyses available to each other. However, both approaches do not detail how analyses can interact to reach an analysis goal. In contrast, we define the specific analysis goals of our coupling and how analyses interact in the specialization of the coupling approach in our coupling approaches.

Simulation is another type of analysis for which analysis coupling has received major attention in research and industry, and for which various frameworks and standards exist. Distributed Interactive Simulation (DIS) [131] and the High-Level Architecture (HLA) [57] are examples of simulation coupling frameworks for co-simulation of different simulations that were initially not designed to work together. The functional mockup interface (FMI) [32] describes a standard for simulation coupling by providing a common interface for simulation units. Ptolemy II [262] provides different models of computation to enable actor-based simulation coupling. FERAL [177] allows describing the information required and provided through ports and directors that trigger simulation components. All of these coupling approaches differ from our contributions, as they conform to grey-box coupling to coordinate simulation steps.

In summary, to the best of our knowledge, there is no approach for coupling architectural analyses and source code analyses to calculate the security characteristic resulting from the implementation based on non-compliance between security characteristics specified in architectural models and the implementation, which is the focus of our coupling design and coupling approaches. There are many approaches for cooperative source code analysis; however, they lack a focus on coupling analyses across different system abstraction levels. Specifically, these approaches focus on enabling cooperation between analyses on the level of source code. We see simulation coupling approaches as related, as they also focus on coupling analyses across different system abstractions. However, the found approaches conform to grey-box coupling to coordinate simulation steps, while our coupling design and coupling approaches conform to black-box coupling. Moreover, coupled simulations are commonly used in the same development phase (e.g., simulating systems in the design phase), while our coupling design and coupling approaches focus on coupling analyses across different development phases (i.e., architectural analyses in the architectural phase and source code analyses in the implementation phase). Consequently, we see our contributions as distinct from related work by focusing on coupling analyses with a black-box coupling approach, maintaining a dedicated security focus, and being applicable across different development phases.

11. Conclusion

In this chapter, we conclude this dissertation. First, we briefly summarize the contributions, their validation and findings we made throughout this work in Section 11.1. Then, we discuss the benefits of our contributions in Section 11.2. Finally, we outline directions for future research in Section 11.3.

Note that each contribution chapter and evaluation chapter contains a detailed summary. Specifically, Section 4.6 summarizes the coupling design (**C1**), Section 5.4 summarizes our coupling approaches (**C2**), and Section 6.5 summarizes *SecLan* which allows relating security DSLs with source code security analyzers (**C3**). Section 7.3 summarizes the evaluation goals and data collection methods applied to evaluate our three contributions **C1**, **C2** and **C3**. Section 8.9 and Section 9.5 summarize the study designs and the evaluation results respectively in the evaluation of the coupling design and coupling approaches, and in the evaluation of *SecLan*.

11.1. Summary

This thesis is positioned at the intersection of security analysis, analysis coupling, and the connection between two system abstractions in software development. The still-rising number of security vulnerabilities in software systems [333] shows the need to support software engineers in detecting vulnerabilities. Approaches such as the Microsoft Security Development Lifecycle [210] highlight this need by requiring security to be ensured in every phase of the development process. However, performing security reviews is laborious and error-prone [77]. This is why various automated security analyses have been proposed for different phases of the development process, such as architectural security analyses for the design phase (e.g., [12, 150, 154, 300, 348]) and static source code analyses for the implementation phase (e.g., [18, 19, 176, 260, 310]).

In this dissertation, we focus on architectural security analyses, a subclass of design-level analyses, that detect architectural vulnerabilities in the system architecture. This focus is reasonable due to the strong dependencies of quality properties such as security on architectural decisions [121]. Architectural analyses commonly use models of the system architecture enriched with security characteristics [299] to detect architectural vulnerabilities.

Ideally, once architectural analyses have determined a system to be free of architectural vulnerabilities, it is implemented according to the system design and security specification.

However, evolutionary changes can lead to non-compliance between the system design and its implementation [159, 337]. In this dissertation, we focus on *specification-related non-compliance*, which occurs when the implementation does not realize the security properties specified in the architectural analysis input model. This non-compliance can prevent architectural analyses from detecting architectural vulnerabilities that manifest in the implemented system, because the security characteristics specified in the architectural analysis input model differ from those resulting from the implementation.

We address this challenge by coupling architectural security analyses with static source code security analyses to enrich the architectural analysis with the values of security characteristics that result from the implementation. This coupling allows us to conduct the architectural analysis with values of security characteristics that result from the implementation rather than security characteristics specified in early design phases. Consequently, the architectural analysis can detect vulnerabilities that arise in the final system which cannot be detected by either analysis in isolation. However, analysis coupling requires determining suitable properties for the coupling, bridging the gap between the vulnerabilities detectable by architectural analyses and source code analyses, and establishing the information exchange between the analyses. In particular, selecting source code analyses capable of detecting non-compliance relevant to the architectural analysis is challenging because the relationship between the source code analysis result and security properties — often described in design-level security DSLs [174] — is not yet systematically captured [367].

We address these challenges by making three contributions in this dissertation: (**C1**) a coupling design that defines properties for coupling architectural security analyses with static source code security analyses, (**C2**) two coupling approaches that instantiate the coupling design, and (**C3**) *SecLan*, a systematic approach to relate security DSLs and static source code security analyses. We use a running example throughout this thesis to illustrate the concepts of our contributions. In the following, we summarize our contributions and their evaluations. Note that we already provide detailed summaries for each contribution and its evaluation in the respective chapters. Section 4.6 summarizes the coupling design, Section 5.4 summarizes the coupling approaches, and Section 6.5 summarizes *SecLan*. The evaluation of the coupling approaches is summarized in Section 8.9, and the evaluation of *SecLan* is summarized in Section 9.5.

After providing summaries of the contributions and evaluations, we discuss whether we reached our research objective stated in Section 1.3.

11.1.1. Summary of Contributions

We now summarize our three contributions (**C1** to **C3**) presented in this dissertation.

Coupling Design (C1) Our first contribution describes the properties that coupling approaches must exhibit to be suitable for coupling architectural security analyses with static

source code security analyses. This contribution is presented in Chapter 4. We detail the *coupling configuration*, which defines the suitability of a sequential black-box coupling with the goal of detecting architectural vulnerabilities arising from an implementation that is non-compliant with the architectural specification. This analysis coupling uses only the input and output models of the analyses and transformations between them to parameterize the architectural analysis with the results of the source code analysis.

We define roles with responsibilities and competencies for the development of coupled analyses to address the complexity of analysis coupling. We also discuss the influence of the semantics of the information in the analyses on the coupling. In particular, we discuss three types of semantic relations (semantically related, semantically identical, semantically unrelated) that affect potential analysis couplings.

As part of this contribution, we provide a coupling process that defines necessary steps to achieve the coupling. In this process, we define the models involved in the coupling and the process steps to be performed. In our coupling, we abstract from specific analyses and their models by using reference metamodels, which define the information necessary in the metamodels of the analyses to perform the coupling. The process steps define transformations between the models of the analyses to achieve the coupling. In the coupling design, we describe primary tasks the transformations must perform.

We state that a single coupling approach for all analyses is infeasible because of the highly diverse types of analyses, security properties, and semantic relations that arise from them. Therefore, a coupling approach must instantiate the coupling design by defining the necessary information and mechanisms to bridge the gap between the analyses. In particular, the coupling process must be specialized by the coupling approach to define how the coupling is achieved for specific types of analyses and security properties.

With contribution **C1** we answer **Research Question 1** and address **Problem 1** by defining the properties of coupling approaches suitable for coupling architectural security analyses with static source code security analyses to detect architectural vulnerabilities arising from non-compliance.

Coupling Approaches (C2) Our second contribution provides two coupling approaches that instantiate the coupling design presented as our first contribution. This contribution is presented in Chapter 5. These approaches illustrate the coupling for different types of static source code analyses, security characteristics used in architectural security analyses, and different semantic relations to be addressed. The first coupling approach — called *specification-based coupling approach* — couples architectural analyses with specification-based source code information flow analyses to investigate architectural vulnerabilities arising from non-compliant information flows. The second coupling approach — called *pattern-search-based coupling approach* — couples architectural security analyses with pattern-based source code analyses to investigate architectural vulnerabilities arising from a non-compliant implementation due to encryption-related vulnerabilities.

For both coupling approaches, we define the reference metamodels necessary to perform the coupling and the conditions under which non-compliance arises in the context of the analyses and security properties addressed by the coupling approach. Furthermore, we define the specialization of the coupling process for each coupling approach, which includes the necessary transformations and mechanisms to bridge the gap between the analyses in the coupling. We also discuss the effort arising in realizing the coupling approaches based on the categories *repeated manual effort* and *one-time effort*.

However, the coupling approaches differ in the semantic relations they address and the mechanisms they use to bridge the gap between the analyses. In particular, the specification-based coupling approach exploits the semantically identical relation type between the input model and output models of the analyses in the coupling, which does not require complex techniques to calculate the relations between the models during the coupling. Thus, we focus on specifying Integration Conditions (ICs) that define necessary but not sufficient relations between elements of the input and output models of the analyses to be coupled. These ICs can be used by the coupling expert to create the transformations necessary to perform the coupling process. Furthermore, we define the necessary tasks for the transformations of the coupling process which also address these ICs. These transformations include, amongst others, the major task of establishing relations that allow calculating the *values resulting from the implementation* from the information flows detected by the source code analysis. In contrast, the pattern-search-based coupling approach addresses the semantically related relation type between the input and output models of the analyses in the coupling, which requires techniques to calculate the relations between the models during the coupling. Thus, we focus on these techniques and the information necessary to perform the coupling and omit defining ICs.

The coupling approaches presented in this contribution answer **Research Question 2** and **Research Question 3** by detailing relations and mechanisms used to bridge the abstraction gap between the analyses. Consequently, they address **Problem 1** by providing concrete coupling approaches that can be applied to detect architectural vulnerabilities arising from non-compliance between architectural specifications and implementations. Furthermore, they address **Problem 2** by providing systematic approaches to relate the vulnerabilities detected by source code analyses to the vulnerabilities detected by architectural analyses. Because the ICs and reference metamodels defined in these coupling approaches can also be used to determine whether two analyses are related, this contribution also partly answers **Research Question 4**; therefore, partly addressing **Problem 3**.

***SecLan* (C3)** Our third contribution is *SecLan*, a systematic approach to relate security DSLs and static source code security analyses. This contribution is presented in Chapter 6. *SecLan* provides the *SecLan* conceptual model, which contains concepts and relations used to relate security DSLs and static source code security analyzers. The *SecLan* conceptual model is synthesized by applying a systematic process that involves investigating 66 security DSLs and 33 static source code security analyzers with 408 security checks to identify concepts common to both domains. The *SecLan* conceptual model comprises four

sub-models: the *Security Model*, the *System Model*, the *Security DSL Description* and the *Security Analyzer Description*. These models contain concepts and relations that allow relating security DSLs and static source code security analyzers. We provide an instance of the *Security Model* which instantiates the STRIDE [304] threat model. Furthermore, we provide a specific *System Model* which captures abstractions of system elements to relate security DSLs and analyzers.

Using these concepts and relations, *SecLan* can determine relationships indicating that a source code security check detects vulnerabilities that potentially violate security properties specified in security DSLs (i.e., non-compliance detection). *SecLan* calculates these relationships by representing instances of the *SecLan* conceptual model describing security DSLs and source code analyzers as a combined graph. A security check is related to a security DSL if a regular path expression over this graph can be traversed to connect the security check and the security DSL. The traversal connects the security DSLs and security checks over the *Security Model* and the *System Model*.

SecLan answers **Research Question 4**, and thus addresses **Problem 3**, by defining the information that allows relating security DSLs and static source code security analyses so that the source code analysis is capable of detecting non-compliance regarding security designs described by the security DSL.

11.1.2. Summary of Evaluation

The contributions presented in this dissertation were validated using quantitative and qualitative methods. We validated the coupling design (**C1**) and coupling approaches (**C2**) jointly because the coupling approaches instantiate the coupling design, while evaluating *SecLan* (**C3**) independently. For structuring the evaluation of our contributions, we discussed the evaluation objectives and suitable data collection methods in Chapter 7. In this section, we structure the quantitative evaluation based on a Goal-Question-Metric (GQM) plan [23]. In this GQM plan, we defined three evaluation goals, 12 evaluation questions, and 8 metrics. For completeness, we also capture the qualitative evaluation in the GQM plan as a dedicated goal, although we do not provide dedicated questions and metrics for this evaluation. In the following, we summarize the evaluation of the coupling design and coupling approaches, as well as the evaluation of *SecLan*, and include information about the goals, questions, metrics, and study design for each evaluation.

Evaluation of Coupling Design and Coupling Approaches The evaluation of the coupling design and coupling approaches is described in Chapter 8. We evaluate (1) the *coverage of the reference metamodels and ICs by the input and output models of the analyses* (**G1**), and (2) the *accuracy of the coupling approaches in detecting architectural vulnerabilities arising from non-compliance* (**G2**). We apply an *evaluation scenario*-based design for this evaluation, which is similar to a case study but highlights that we do not use real-world cases. Each evaluation scenario comprises a *coupling scenario* (i.e., an architectural analysis, a source

code analysis and a security characteristic to be analyzed for non-compliance) and a *system* to be analyzed.

Due to the different requirements for the ground truth necessary to evaluate the accuracy of both coupling approaches, we apply two separate study designs for their evaluation. To the best of our knowledge, there is no comprehensive ground truth in related work that describes architectural vulnerabilities arising from non-compliance in implementations of several systems. Therefore, in our study design, we created architectural models, security specifications, and corresponding implementations. Furthermore, we injected vulnerabilities into the implementations that result in non-compliance. For the definition of the ground truth, we assumed that the source code analyses are accurate in detecting the vulnerabilities injected into the implementation.

For evaluating the *specification-based coupling approach*, we created *sixteen* evaluation scenarios by coupling *two* architectural analyses with *two* specification-based source code analyses each for *one* security characteristic, and applying this coupling to *four* systems. For creating the ground truth, we injected information flows that resulted in non-compliance into the systems. The injected information flows are based on three equivalence classes that influence the coupling. For evaluating the *pattern-search-based coupling approach*, we created *nine* evaluation scenarios by coupling *one* architectural analysis with *three* pattern-based source code analyses individually for *one* security characteristic, and applying these couplings to *three* systems. We injected encryption-related vulnerabilities into the systems to create the ground truth. These vulnerabilities were based on three equivalence classes that are intended to illustrate different challenges for the coupling approach.

Overall, the evaluation showed that the reference metamodels define interfaces for the coupling and that the ICs are indeed necessary conditions. Our evaluation of accuracy showed that both coupling approaches are capable of detecting architectural vulnerabilities arising from non-compliance in most cases. However, the accuracy depends strongly on the realized coupling approach and the analyzed scenario. While we found in the evaluation of the *specification-based coupling approach* 60 true positives, 5 false positives and 0 false negatives, the evaluation of the *pattern-search-based coupling approach* resulted in 15 true positives, 3 false positives and 9 false negatives. The findings indicate that we need to further improve the pattern-search-based coupling approach to increase its accuracy.

We also discuss limitations and threats to validity of our evaluation in Section 8.7. We discuss the threats to validity structured by the categories *Internal Validity*, *External Validity*, *Construct Validity*, and *Reliability* of Runeson and Höst [282].

Evaluation of *SecLan* The evaluation of *SecLan* is described in Chapter 9. We (1) evaluate the *correctness and completeness of the* SecLan *conceptual model and its instances* (**G3**), and (2) qualitatively explore the relationships calculated by *SecLan* (**G4**). The correctness and completeness of the *SecLan* conceptual model must be evaluated to assess whether it can capture the domains of security DSLs and static source code analyzers correctly and completely. This is necessary because the *SecLan* conceptual model and its instances form the basis for calculating relationships between security DSLs and static source code

analyzers. We cannot apply quantitative methods to evaluate the relationships calculated by *SecLan* because no comprehensive ground truth is available and because we aim to gather insights about relationships that we cannot anticipate. Consequently, we qualitatively explored the calculated relationships with experts to gather insights about their correctness and obtain feedback we could not anticipate.

We cannot apply an evaluation scenario-based design for this evaluation because a ground truth created by us would be affected by biases and misinterpretations. Therefore, we conducted surveys and interviews with experts in security DSLs and static source code security analyzers, which are established techniques in the literature (e.g., [15]) for validating conceptual models. We invited a total of 132 authors of research papers regarding security DSLs and static source code security analyzers and also maintainers of these tools as experts for our evaluation. From these invited experts, 17 participated in the survey and 7 participated in the interviews. We use the survey to evaluate the correctness and completeness of the *SecLan* conceptual model and its instances. This survey first presents the experts with the concepts and relations between the concepts from the *SecLan* conceptual model as well as the provided *Element Types* and the relations between them from the *System Model*. For every sub-model, experts are expected to assess whether the concepts and their relations, or the provided *Element Types*, fit their understanding. Furthermore, the experts are asked to assess the correctness and completeness of the provided *SecLan* conceptual model instances for specific security DSLs and static source code security analyzers of their choice.

The survey results show that the experts rated the *SecLan* conceptual model and its instances as mostly correct and complete. In particular, the feedback can result in a total of 863 agreements or disagreements with the elements of the *SecLan* conceptual model. In the survey and interview, only 7 disagreements were raised, resulting in a ratio of $\frac{7}{863} = 0.008$. Still, we made some improvements to the *SecLan* conceptual model, and found challenges in modeling one security DSL due to similarities to *SecLan*.

We use the interviews mainly to gather qualitative insights into the relationships calculated by *SecLan*. However, first, we also again assess the correctness of the *SecLan* conceptual model and its instances with the interview participants. We gather this information to detect deviations between the understanding of the experts regarding the *SecLan* conceptual model in the survey and during the interviews. This allows us to detect maturation effects. Furthermore, we also assess the correctness and completeness of the instances of the *SecLan* conceptual model with the interview participants to detect issues we could have introduced when modifying them based on the survey feedback.

The results of the interviews showed that most relationships were rated as correct by the experts. In particular, the experts highlighted the benefit of the reasoning provided by *SecLan* to understand the rationale behind the relationships. This capability was also shown because most experts initially flagged some relationships as incorrect, but changed their assessment after investigating the reasoning provided by *SecLan*. The greatest room for improvement was identified by the experts in the large number of relationships provided by *SecLan*, which can overwhelm the coupling expert using *SecLan*.

Again, we discussed limitations and threats to validity of our evaluation in Section 9.3. Similar to the evaluation of the coupling design and coupling approaches, we discussed the threats to validity structured by the categories *Internal Validity*, *External Validity*, *Construct Validity*, and *Reliability* of Runeson and Höst [282].

11.1.3. Discussing the Research Objective

In this dissertation, we formulated the research goal in Section 1.3 that we aim to enable software engineers to investigate the impact of non-compliance between architectural security design and implementations by coupling static source code analyses with architectural security analyses. For this purpose, we proposed to couple architectural security analyses with static source code security analyses. We aimed to (1) specify properties that constitute an analysis coupling appropriate for coupling architectural security analyses and static source code security analyses, (2) provide approaches for relating implementation vulnerabilities detected with source code analyses that imply non-compliance to architectural security, and (3) provide an approach for selecting static source code analyses capable of detecting non-compliance that affects the architectural security investigated by an architectural analysis.

We addressed this research objective by providing a coupling design (**C1**) that structures the coupling of architectural security analyses with static source code analyses. Based on this coupling design, we provide two coupling approaches (**C2**) that realize the coupling design for different types of static source code analyses and architectural security analyses. Finally, we provide *SecLan* (**C3**) to systematically identify relationships between security DSLs and static source code analyses to support coupling experts in selecting analyses for the coupling.

The evaluation results of the coupling design and coupling approaches show that we provide a coupling design that allows us to couple architectural security analyses with static source code analyses while accounting for the needs arising from the different system abstractions. Furthermore, the evaluation shows that the coupling approaches are capable of detecting architectural vulnerabilities arising from non-compliance. Finally, the evaluation of *SecLan* shows that we provide a systematic approach to relate security DSLs and static source code analyses.

Based on the evidence provided by our evaluation, we conclude that we provide approaches that address the research objective of this dissertation. However, we also found challenges and limitations in our approaches that have to be addressed in future work to further improve how well we can address this research objective. We will discuss such future work in Section 11.3.

11.2. Discussion of Benefits

We see six benefits arising from the contributions presented in this dissertation. These benefits are sequentially denoted as **B1** to **B6** and are described in the following.

Regarding our coupling design (**C1**) and coupling approaches (**C2**), we see three benefits (**B1**, **B2**, **B3**). The coupling design provides reusable knowledge on properties suitable to couple architectural analyses with source code analyses (**B1**). This claim is supported by the fact that we could successfully instantiate two coupling approaches based on this design to couple architectural analyses with two major types of source code analyses. These coupling approaches provide researchers and software engineers with practical approaches to detect security vulnerabilities arising from non-compliant implementations.

We also claim two benefits of the analysis coupling proposed in this dissertation. First, software engineers become aware of vulnerabilities arising from non-compliant implementations, which is not possible when applying the architectural analysis in isolation (**B2**). This information is crucial for software engineers to be able to fix security vulnerabilities in their systems before they can be exploited by attackers. This benefit has been shown in our evaluation of the coupling approaches by detecting architectural vulnerabilities arising from non-compliant implementations that would have been missed without the coupling.

Secondly, software engineers can distinguish whether a reported implementation vulnerability leads to an architectural vulnerability (**B2**). With this capability, software engineers have additional information to prioritize which vulnerabilities to fix, which is an important task in practice [203] because achieving full security is rarely possible [126]. We illustrate this capability through the class *no effect* in our evaluation. We also illustrate **B3** using our running example (see Chapter 3) in the following. In our running example, the information flow analysis would report two information flows that violate the security policy in the implementation: (1) from the parameter *userData* in the method *buy* of the class *OnlineShop* to the parameter *userData* of the method *store* in the class *DataStorage*, and (2) from the parameter *userData* in the method *buy* of the class *OnlineShop* to the parameter *entry* of the method *log* in the class *Logger*. Both information flows carry confidential information of *userData* to locations expected to contain non-confidential data. Because both information flows carry the same information, they can be assumed to be equally important vulnerabilities. However, when applying our analysis coupling, software engineers can obtain insight that only the first information flow leads to an architectural vulnerability whereas the second does not. This insight allows software engineers to decide which vulnerability to fix first. Note that we do not claim that only those vulnerabilities that lead to architectural vulnerabilities should be fixed first. However, software engineers can make more informed decisions about which vulnerabilities to fix first based on the information about whether a vulnerability leads to an architectural vulnerability or not.

Our evaluation also uncovered two benefits of *SecLan* (**B4** and **B5**). With *SecLan*, we provide software engineers with a systematic approach to identify source code analyses capable of

detecting security vulnerabilities that violate security-related properties designed in design-time security DSLs (**B4**). We have shown this capability in our qualitative exploration of the relationships calculated by *SecLan* with security experts. Thus, *SecLan* can help software engineers find suitable source code analyses to detect security vulnerabilities in their systems that affect their security design. This capability of *SecLan* is especially important as our exploration showed that even security experts are not aware of many relationships between security properties specified in security DSLs and the vulnerabilities detected by source code analyses. *SecLan* also helps coupling experts find suitable source code analyses for our analysis coupling, because the architectural analysis input model is commonly described with security DSLs, which serve as primary artifacts used in the coupling. Furthermore, our exploration found evidence that *SecLan* can feasibly provide *reasoning* about *why* certain source code analyses are related to security DSLs (**B5**), which is otherwise not always obvious to security experts. This reasoning can help software engineers to better understand the relationships between security properties designed in design-time security DSLs and the vulnerabilities detected by source code analyses.

Finally, in this thesis, we see one benefit of all our contributions for research and practice: We take a further step to address the challenge of bridging the gap between architectural security and implementation security, as well as the compositionality of security properties (**B6**). With *SecLan*, we systematically bridge the gap between security properties designed in design-time security DSLs and vulnerabilities detected by source code analyses. In the coupling design and coupling approaches, we bridge the gap between vulnerabilities in the implementation and vulnerabilities in the architecture. Providing insights into this gap is crucial to tackle challenges like architectural erosion [62] and to ensure that security properties designed at the architectural level are also fulfilled in the implementation. Furthermore, we also address the challenge that the compositionality of security properties is not yet well understood [71] by providing insights into how vulnerabilities in the implementation can lead to vulnerabilities in the architecture. Researchers can build upon our contributions to further address these challenges, for example by relating findings from source code analyses and architectural analyses to software requirements.

11.3. Future Work

Our contributions in this thesis present a step further towards engineering secure software systems by combining information from the architecture and the implementation. However, this thesis is far from solving all challenges in this domain and leaves several opportunities for future work, which are discussed in this section.

We classify this future work into (1) *Automation and Constructive Guidance*, (2) *Capability Usage, Adaptation and Generalization*, and (3) *Evaluation*. *Automation and Constructive Guidance* comprises future work that provides tool support or guidance to perform tasks in the coupling process or in *SecLan*. *Capability Usage, Adaptation and Generalization* comprises future work that uses our contributions in other contexts, adapts our contributions to perform similar capabilities or generalizes the contributions to other scenarios.

Evaluation comprises future work that extends the evaluation of our contributions to other quality properties or to address threats to validity.

We consecutively enumerate the future work with **FW1** to **FW24**. Note that this list is not complete because we only began to explore the research field of coupling architectural analyses and source code analyses for security engineering.

Automation and Constructive Guidance In the coupling design and coupling approaches, the limitations **L1**, **L2**, **L4** (see Section 4.5), **L8** and **L9** (see Section 5.3) highlight that we assign several tasks to the *coupling expert* that have to be performed manually. Consequently, these tasks are prone to errors and can result in reduced quality of the coupling. Moreover, while we raise awareness of required tasks in the coupling and provide fundamental insights into how to perform them, we do not include detailed guidance or tool support for these tasks. Currently, the *coupling expert* must manually determine the *semantic relations* to select an appropriate coupling approach (**L4**). This task can be supported by further research in the domain of *ontology matching* [306] (**FW1**). With the rise of Large Language Models (LLMs), future work can investigate whether a LLM can support the coupling expert in determining the *semantic relations* between architectural elements and implementation elements (**FW2**). The architectural vulnerabilities arising from non-compliance must be related to the sources of non-compliance in the implementation to guide software engineers in prioritizing fixes in the implementation. However, we do not investigate how to establish this relation (**L2**). Thus, future work can comprise extending the coupling so that it establishes traces between architectural vulnerabilities and sources of non-compliance in the implementation (**FW3**).

The pattern-search-based coupling approach requires manually providing the *Security Relations* model (**L9**). Similarly, the accuracy and explainability of relations calculated by *SecLan* suffer from *Security Checks* being mapped to rather abstract *Weaknesses*, as found in our qualitative exploration (see Subsection 9.2.2). For both limitations, future work can investigate how to provide tool support to define appropriate relations. In the context of the pattern-search-based coupling approach, we performed an initial investigation on using LLMs to find weaknesses that violate a security characteristic, and create the *Security Relations* model. Thus, we propose research investigating to what extent LLMs can support the creation of the *Security Relations* model (**FW4**) or the mapping of *Security Checks* to appropriate *Weaknesses* (**FW5**).

In the presented coupling approaches, the calculation of *values resulting from the implementation* must be implemented manually (**L8**). Preliminary research in the master's thesis of Lehmann [185] shows that different semantics of security characteristics of data can require distinct ways of calculating these values or handling them in the analysis coupling. Consequently, future work can build on these results to provide more guidance on how to calculate *values resulting from the implementation* more precisely for different semantics of security characteristics of data (**FW6**). Furthermore, the challenge arising from not considering semantics of implementation elements in the pattern-search-based coupling approach (**L10**) is one factor that influences the accuracy of the coupling as

illustrated in our evaluation. Future work can investigate how to incorporate semantics of methods into coupling model to avoid the need for approximating this information in the search strategies (**FW7**). An initial approach for this future work can be to use additional information sources like SecDFDs in the approach of Peldszus et al. [256].

Capability Usage, Adaptation and Generalization In this dissertation, we focus on static source code analyses to detect non-compliance (**L3**) and omit other types such as dynamic analyses. However, this limitation does not allow us to detect all types of non-compliance that could lead to architectural vulnerabilities. Thus, future work can investigate how to incorporate other types of analyses, such as dynamic analysis, into the coupling design, coupling approaches and *SecLan* (**FW8**).

The coupling design and coupling approaches currently only detect architectural vulnerabilities arising from non-compliance between the architecture and implementation (**L5**). We propose to investigate how the coupling design and the coupling approaches can be extended to evaluate whether architectural vulnerabilities are removed due to the implementation despite being non-compliant (**FW9**). In this line, we also propose to investigate not only how to remove annotations or change their values, but also how to *add* annotations to the architectural model based on the implementation (**FW10**), e.g., when encryption is used in the implementation although not specified in the architecture.

Our coupling design is currently limited by only using the structural descriptions of the architectural model (**L6**). This limitation results in not addressing design-level analyses with behavioral models like data flow analyses (e.g., SecDFD [338]). Thus, future work can investigate how to adapt the coupling design to incorporate behavioral models in addition to structural models (**FW11**).

Similarly, in our coupling approaches we focus on non-compliance arising from vulnerabilities originating from information flows or encryption-related errors in the implementation (**L7**). Although our coupling process, the ICs and reference metamodels are not tailored to these vulnerabilities, we cannot guarantee that our coupling approaches generalize to other types of analyses, security characteristics and vulnerabilities. Thus, future work can investigate whether our coupling design and coupling approaches can be generalized to other types of analyses, security characteristics and vulnerabilities (**FW12**).

Currently, *SecLan* does not incorporate security characteristics from architectural models (**L14**). This leads to a gap between the information available in *SecLan* and the information required for coupling architectural analyses and source code analyses. Consequently, we propose to investigate how to directly relate the findings of *SecLan* to the analysis coupling proposed in this dissertation (**FW13**). We performed a systematic process based on the selected data sources to synthesize the *SecLan* conceptual model (see Section 6.1) (**L12**). However, we do not claim completeness of these sources. Thus, we propose to investigate further data sources to extend the *SecLan* conceptual model (**FW14**).

Security issue prioritization is a common task in practice [203]. We already discuss future work in **FW3** to support prioritization in the coupling by tracing architectural vulner-

abilities to sources of non-compliance in the implementation. We propose extending prioritization of security vulnerabilities across other system abstractions or security requirements based on insights from the analysis coupling (**FW15**). For this, we see strong synergy with the work of Peldszus et al. [253] which prioritizes source code analysis results regarding stated requirements. In this work, our analysis coupling can be applied to consider whether the source code vulnerabilities result in architectural vulnerabilities.

Finally, LLMs are increasingly used in source code generation [149, 350]. However, some research [85, 307] suggests that LLMs can introduce security vulnerabilities into the generated source code. Thus, we propose future work to investigate how to use principles of our analysis coupling — such as generating specifications for source code analysis — to analyze source code generated by LLMs and to provide feedback to the LLM to avoid generating insecure source code (**FW16**).

Evaluation Our evaluations are affected by several limitations and threats. First, the study designs of our coupling design and coupling approaches are affected by the limited number of systems, architectural analyses and source code analyses (**L16** and **L17**). This limitation is especially evident for the study design applying the pattern-search-based coupling approach. Therefore, we propose future work by applying our coupling approaches to other architectural analyses, source code analyses, and systems (**FW17**).

We also discuss that we only use a subset of semantics of values of security characteristics of data in the study design applying the specification-based coupling approach; namely roles with *disjunctive* lattices, and confidentiality levels (values *high* and *low*) with strictly ordered lattices (**L18**). Thus, we propose future work to extend the evaluation of the specification-based coupling approach by applying further semantics of values of security characteristics of data such as roles with *conjunctive* lattices or arbitrary security lattices (**FW18**).

Furthermore, we only use one coupling model in the pattern-search-based coupling approach (**L19**). Thus, we do not evaluate the effect of using other coupling model on the quality of the coupling. Although we provide a first investigation in the bachelor's thesis of Traub [336], we propose future work to extend this investigation by applying other coupling model such as system dependence graphs [347] or the GRaViTY program model [250] (**FW19**).

One further limitation is that we inject only a limited number of vulnerabilities into the systems; especially in the study design applying the pattern-search-based coupling approach (**L20**). This is especially pertinent because many of the injected vulnerabilities are based on rationale about their plausibility instead of existing descriptions. Thus, we propose to find systems for which descriptions of existing vulnerabilities are available to extend the evaluation of the pattern-search-based coupling approach (**FW20**).

We also only evaluate a limited number of quality attributes in our evaluation of the coupling design and coupling approaches (**L21**). Although we evaluate *coverage* and *accuracy*, we omit other quality attributes such as *scalability* or *effort*. Therefore, we propose

future work to extend the evaluation of the coupling design and coupling approaches by evaluating further quality attributes (**FW21**). We want to highlight evaluating *scalability* and *effort* as especially relevant future work as they are crucial for practical applicability. While we discussed the effort in Subsection 5.1.5 and Subsection 5.2.6 argumentatively, an empirical evaluation would be beneficial to obtain stronger evidence.

The validity of the evaluation of *SecLan* is mainly threatened by the limited number of participants in the expert survey and interviews (**L22**). We found current email addresses of 132 experts, which we invited to participate in the survey and interviews. However, only 17 experts participated in the survey and seven participated in the interviews. We propose future work to extend the evaluation of *SecLan* by involving more experts (**FW22**), e.g., by using incentives or by broadening the scope of invited experts.

Along this line, our evaluation validated only a subset of the created *SecLan* conceptual model instances with experts (**L23**) and relationships (**L24**). Thus, we propose to again extend the evaluation of *SecLan* by validating more *SecLan* conceptual model instances with experts (**FW23**). We also propose future work to explore more relationships calculated by *SecLan* (**FW24**), e.g., by asking experts to explore relationships in a follow-up interview or independently.

Bibliography

The titles of most entries are hyperlinks resolving the DOIs or pointing to other online sources for the entries.

[1] M. Abi-Antoun, D. Wang, and P. Torr. "Checking Threat Modeling Data Flow Diagrams for Implementation Conformance and Security". In: *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*. Ase '07. Association for Computing Machinery, 2007, pp. 393–396.

[2] J. Abramov, A. Sturm, and P. Shoval. "Evaluation of the Pattern-based method for Secure Development (PbSD): A controlled experiment". In: *Information and Software Technology* 54.9 (2012), pp. 1029–1043.

[3] M. Adach, K. Hänninen, and K. Lundqvist. "Security Ontologies: A Systematic Literature Review". In: *Enterprise Design, Operations, and Computing*. Springer International Publishing, 2022, pp. 36–53.

[4] A. Aggarwal and P. Jalote. "Integrating Static and Dynamic Analysis for Detecting Vulnerabilities". In: *30th Annual International Computer Software and Applications Conference (COMPSAC'06)*. Vol. 1. 2006, pp. 343–350.

[5] G.-J. Ahn, S.-P. Hong, and M. E. Shin. "Reconstructing a formal security model". In: *Information and Software Technology* 44.11 (2002), pp. 649–657.

[6] G.-J. Ahn and H. Hu. "Towards realizing a formal RBAC model in real systems". In: *Proceedings of the 12th ACM Symposium on Access Control Models and Technologies*. SACMAT '07. Association for Computing Machinery, 2007, pp. 215–224.

[7] *Deductive Software Verification - The KeY Book - From Theory to Practice*. Vol. 10001. Lecture Notes in Computer Science. Springer, 2016. published.

[8] M. Alam, R. Breu, and M. Hafner. "Model-Driven Security Engineering for Trust Management in SECTET". In: *Journal of Software* 2.1 (2007).

[9] H. H. AlBreiki and Q. H. Mahmoud. "Evaluation of static analysis tools for software security". In: *2014 10th International Conference on Innovations in Information Technology (IIT)*. 2014, pp. 93–98.

[10] J. H. Allen, S. Barnum, R. J. Ellison, G. McGraw, and N. R. Mead. *Software Security Engineering*. Pearson Education, 2008.

[11] M. Almorsy and J. Grundy. "SecDSVL: A Domain-Specific Visual Language to Support Enterprise Security Modelling". In: *2014 23rd Australian Software Engineering Conference*. 2014, pp. 152–161.

[12] M. Almorsy, J. Grundy, and A. S. Ibrahim. “Automated software architecture security risk analysis using formalized signatures”. In: *2013 35th International Conference on Software Engineering (ICSE)*. 2013, pp. 662–671.

[13] A. Alnaim, A. Alwakeel, and E. B. Fernandez. “A Misuse Pattern for Compromising VMs via Virtual Machine Escape in NFV”. In: *Proceedings of the 14th International Conference on Availability, Reliability and Security*. ARES ’19. Association for Computing Machinery, 2019.

[14] A. S. Ami, K. Moran, D. Poshyvanyk, and A. Nadkarni. “"False negative - that one is going to kill you": Understanding Industry Perspectives of Static Analysis based Security Testing”. In: *2024 IEEE Symposium on Security and Privacy (SP)*. 2024, pp. 3979–3997.

[15] S. Ananieva, S. Greiner, T. Kehrer, J. Krüger, T. Kühn, L. Linsbauer, S. Grüner, A. Koziolek, H. Lönn, S. Ramesh, and R. Reussner. “A Conceptual Model for Unifying Variability in Space and Time: Rationale, Validation, and Illustrative Applications”. In: *Empirical Software Engineering* 27.5 (2022).

[16] P. Anderson, L. Kot, N. Gilmore, and D. Vitek. “SARIF-enabled Tooling to Encourage Gradual Technical Debt Reduction”. In: *2019 IEEE/ACM Int. Conf. on Technical Debt (TechDebt)*, pp. 71–72.

[17] L. Apvrille and Y. Roudier. “SysML-Sec: A SysML environment for the design and development of secure embedded systems”. In: *APCOSEC, Asia-Pacific Council on Systems Engineering* (2013), pp. 8–11.

[18] P. Arteau. *Find Security Bugs*. URL: `https://find-sec-bugs.github.io/` (visited on 02/09/2026).

[19] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. “FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps”. In: *SIGPLAN Not.* 49.6 (2014), pp. 259–269.

[20] M. Balliu, D. Schoepe, and A. Sabelfeld. “We Are Family: Relating Information-Flow Trackers”. In: *Computer Security – ESORICS 2017*. Springer International Publishing, 2017, pp. 124–145.

[21] E. Barker. *Guideline for Using Cryptographic Standards in the Federal Government:: Cryptographic Mechanisms*. Tech. rep. 2020.

[22] E. Barker and A. Roginsky. *Transitioning the Use of Cryptographic Algorithms and Key Lengths*. Tech. rep. 2019.

[23] V. R. Basili, G. Caldiera, and H. D. Rombach. “The Goal Question Metric Approach”. In: *Encyclopedia of software engineering* (1994), p. 10.

[24] S. Becker. “Coupled Model Transformations for QoS Enabled Component-Based Software Design”. PhD thesis. Universität Oldenburg, 2008.

[25] D. Bell. “Looking back at the Bell-La Padula model”. In: *21st Annual Computer Security Applications Conference (ACSAC’05)*. 2005, 15 pp.–351.

[26] A. van den Berghe, R. Scandariato, K. Yskout, and W. Joosen. "Design notations for secure software: a systematic literature review". In: *Software & Systems Modeling* 16.3 (2017), pp. 809–831.

[27] S. Bernardi, M. Famelis, J.-M. Jézéquel, R. Mirandola, D. P. Palacin, F. A. C. Polack, and C. Trubiani. "Living with Uncertainty in Model-Based Development". In: *Composing Model-Based Analysis Tools*. Springer International Publishing, 2021, pp. 159–185.

[28] D. M. Berry. "Empirical Evaluation of Tools for Hairy Requirements Engineering Tasks". In: *Empirical Software Engineering* 26.6 (2021).

[29] M. F. Bertoa and A. Vallecillo. "Quality Attributes for Software Metamodels". In: *Proceedings of the 13th TOOLS Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE 2010)*. 2010.

[30] P. E. Black, M. Kass, and M. Koo. *Source Code Security Analysis Tool Functional Specification Version 1.0*. 2007.

[31] P. E. Black, L. Badger, B. Guttman, and E. Fong. *Dramatically Reducing Software Vulnerabilities: Report to the White House Office of Science and Technology Policy*. NIST IR 8151. National Institute of Standards and Technology, 2016.

[32] T. Blochwitz, M. Otter, M. Arnold, C. Bausch, C. Clauß, H. Elmqvist, A. Junghanns, J. Mauss, M. Monteiro, T. Neidhold, et al. "The Functional Mockup Interface for Tool Independent Exchange of Simulation Models". In: *Proceedings of the 8th International Modelica Conference*. Linköping University Press, 2011, pp. 105–114.

[33] B. W. Boehm, R. K. McClean, and D. B. Urfrig. "Some experience with automated aids to the design of large-scale reliable software". In: *Proceedings of the International Conference on Reliable Software*. Association for Computing Machinery, 1975, pp. 105–113.

[34] N. Boltz, S. Hahner, C. Gerking, and R. Heinrich. "An Extensible Framework for Architecture-Based Data Flow Analysis for Information Security". In: *Software Architecture. ECSA 2023 Tracks, Workshops, and Doctoral Symposium*. Springer Nature Switzerland, 2024, pp. 342–358.

[35] M. Brambilla, J. Cabot, and M. Wimmer. *Model-Driven Software Engineering in Practice*. Morgan & Claypool Publishers, 2017.

[36] L. Brent, N. Grech, S. Lagouvardos, B. Scholz, and Y. Smaragdakis. "Ethainter: a smart contract security analyzer for composite vulnerabilities". In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2020. Association for Computing Machinery, 2020, pp. 454–469.

[37] N. Broberg, B. van Delft, and D. Sands. "Paragon for Practical Programming with Information-Flow Control". In: *Programming Languages and Systems*. Springer International Publishing, 2013, pp. 217–232.

[38] J. Brunel, D. Chemouil, L. Rioux, M. Bakkali, and F. Vallée. "A Viewpoint-Based Approach for Formal Safety & Security Assessment of System Architectures". In: *Proceedings of the 11th Workshop on Model-Driven Engineering, Verification and Validation co-located with 17th International Conference on Model Driven Engineering Languages and Systems (MODELS 2014)*. Vol. 1235. 2014, pp. 39–48.

[39] A. Bucaioni, A. Di Salle, L. Iovino, L. Mariani, and P. Pelliccione. "Continuous Conformance of Software Architectures". In: *2024 IEEE 21st International Conference on Software Architecture (ICSA)*. 2024, pp. 112–122.

[40] A. Bucaioni, A. Di Salle, L. Iovino, P. Pelliccione, and F. Raimondi. "Architecture as Code". In: *2025 IEEE 22nd International Conference on Software Architecture (ICSA)*. 2025, pp. 187–198.

[41] M. Buchholtz, S. Gilmore, V. Haenel, and C. Montangero. "End-to-End Integrated Security and Performance Analysis on the DEGAS Choreographer Platform". In: *FM 2005: Formal Methods*. Springer Berlin Heidelberg, 2005, pp. 286–301.

[42] E. Burger. "Flexible Views for View-based Model-driven Development". PhD thesis. 2014. 324 pp.

[43] C. Burt, B. Bryant, R. Raje, A. Olson, and M. Auguston. "Model driven security: unification of authorization models for fine-grain access control". In: *Seventh IEEE International Enterprise Distributed Object Computing Conference, 2003. Proceedings.* 2003, pp. 159–171.

[44] M. Busch, N. Koch, and S. Suppan. "Modeling Security Features of Web Applications". In: *Engineering Secure Future Internet Services and Systems: Current Research*. Springer International Publishing, 2014, pp. 119–139.

[45] C. Calcagno, D. Distefano, J. Dubreil, D. Gabi, P. Hooimeijer, M. Luca, P. O'Hearn, I. Papakonstantinou, J. Purbrick, and D. Rodriguez. "Moving Fast with Software Verification". In: *NASA Formal Methods*. Springer International Publishing, 2015, pp. 3–11.

[46] A. J. Cartwright. "The Elephant in the Room: Cybersecurity in Healthcare". In: *Journal of Clinical Monitoring and Computing* 37.5 (1, 2023), pp. 1123–1132.

[47] F. Cheirdari and G. Karabatis. "Analyzing False Positive Source Code Vulnerabilities Using Static Analysis Tools". In: *2018 IEEE International Conference on Big Data (Big Data)*. 2018, pp. 4782–4788.

[48] L. Cheng, F. Liu, and D. Yao. "Enterprise Data Breach: Causes, Challenges, Prevention, and Future Directions". In: *WIREs Data Mining and Knowledge Discovery* 7.5 (2017), e1211.

[49] B. Chess and J. West. *Secure programming with static analysis*. First. Addison-Wesley Professional, 2007.

[50] M. Clavel, V. da Silva, C. Braga, and M. Egea. "Model-Driven Security in Practice: An Industrial Experience". In: *Model Driven Architecture – Foundations and Applications*. Springer Berlin Heidelberg, 2008, pp. 326–337.

[51] P. Clements. “A survey of architecture description languages”. In: *Proceedings of the 8th International Workshop on Software Specification and Design*. 1996, pp. 16–25.

[52] *CogniCrypt Website*. 2024. URL: https://eclipse.dev/cognicrypt/ (visited on 02/09/2026).

[53] M. Conti, N. Dragoni, and V. Lesyk. “A Survey of Man in the Middle Attacks”. In: *IEEE Communications Surveys & Tutorials* 18.3 (2016), pp. 2027–2051.

[54] V. Cortellessa, R. Eramo, and M. Tucci. “From software architecture to analysis models and back: Model-driven refactoring aimed at availability improvement”. In: *Information and Software Technology* 127 (2020), p. 106362.

[55] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. “StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks”. In: *USENIX Security Symposium*. 1998.

[56] S. Cruanes, G. Hamon, S. Owre, and N. Shankar. “Tool Integration with the Evidential Tool Bus”. In: *Verification, Model Checking, and Abstract Interpretation*. Springer Berlin Heidelberg, 2013, pp. 275–294.

[57] J. S. Dahmann, R. M. Fujimoto, and R. M. Weatherly. “The Department of Defense High Level Architecture”. In: *Proceedings of the 29th Conference on Winter Simulation*. WSC ’97. IEEE Computer Society, 1997, pp. 142–149.

[58] L. Dai and K. Cooper. “Modeling and performance analysis for security aspects”. In: *Science of Computer Programming* 61.1 (2006). Special Issue on Quality system and software architectures, pp. 58–71.

[59] P. Dalgaard. *Introductory Statistics with R*. Springer New York, 2008.

[60] N. Daswani and M. Elbayadi. “The Equifax Breach”. In: *Big Breaches: Cybersecurity Lessons for Everyone*. Apress, 2021, pp. 75–95.

[61] I. David, H. Vangheluwe, and E. Syriani. “Model consistency as a heuristic for eventual correctness”. In: *Journal of Computer Languages* 76 (2023), p. 101223.

[62] L. de Silva and D. Balasubramaniam. “Controlling software architecture erosion: A survey”. In: *Journal of Systems and Software* 85.1 (2012). Dynamic Analysis and Testing of Embedded Software, pp. 132–151.

[63] D. E. Denning. “A lattice model of secure information flow”. In: *Commun. ACM* 19.5 (1976), pp. 236–243.

[64] P. Diaz, I. Aedo, D. Sanz, and A. Malizia. “A model-driven approach for the visual specification of Role-Based Access Control policies in web systems”. In: *2008 IEEE Symposium on Visual Languages and Human-Centric Computing*. 2008, pp. 203–210.

[65] G. Díaz and J. R. Bermejo. “Static analysis of source code security: Assessment of tools against SAMATE tests”. In: *Information and Software Technology* 55.8 (2013), pp. 1462–1476.

[66] J. Díaz-Pace, Á. Soria, G. Rodríguez, and M. R. Campo. “Assisting conformance checks between architectural scenarios and implementation”. In: *Information and Software Technology* 54.5 (2012), pp. 448–466.

[67] C. Dietrich, K. Krombholz, K. Borgolte, and T. Fiebig. "Investigating System Operators' Perspective on Security Misconfigurations". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. CCS '18. Association for Computing Machinery, 2018, pp. 1272–1289.

[68] A. Djoudi, M. Hána, and N. Kosmatov. "Formal Verification of a JavaCard Virtual Machine with Frama-C". In: *Formal Methods*. Springer International Publishing, 2021, pp. 427–444.

[69] F. Durán, M. Gogolla, E. Guerra, J. d. Lara, H. Sahraoui, and S. Zschaler. "Exploiting Results of Model-Based Analysis Tools". In: *Composing Model-Based Analysis Tools*. Springer International Publishing, 2021, pp. 129–158.

[70] F. Durán, R. Heinrich, C. Talcott, and S. Zschaler. "Conclusion". In: *Composing Model-Based Analysis Tools*. Springer International Publishing, 2021, pp. 301–307.

[71] F. Durán, R. Heinrich, C. Talcott, and S. Zschaler. "Overview of Challenges in Composing Model-Based Analysis Tools". In: *Composing Model-Based Analysis Tools*. Springer International Publishing, 2021, pp. 41–43.

[72] J. Düsing and B. Hermann. "Persisting and Reusing Results of Static Program Analyses on a Large Scale". In: *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2023, pp. 888–900.

[73] M. B. Dwyer and S. Elbaum. "Unifying verification and validation techniques: relating behavior and properties through partial evidence". In: *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*. FoSER '10. Association for Computing Machinery, 2010, pp. 93–98.

[74] J. Ebert and D. Bildhauer. "Reverse Engineering Using Graph Queries". In: *Graph Transformations and Model-Driven Engineering: Essays Dedicated to Manfred Nagl on the Occasion of his 65th Birthday*. Springer Berlin Heidelberg, 2010, pp. 335–362.

[75] M. Eby, J. Werner, G. Karsai, and A. Ledeczi. "Integrating Security Modeling into Embedded System Design". In: *14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'07)*. 2007, pp. 221–228.

[76] Eclipse Foundation. *Eclipse Secure Storage*. URL: `https://help.eclipse.org/latest/topic/org.eclipse.platform.doc.isv/guide/secure_storage.htm?cp=2_0_3_7_0` (visited on 02/09/2026).

[77] A. Edmundson, B. Holtkamp, E. Rivera, M. Finifter, A. Mettler, and D. Wagner. "An Empirical Study on the Effectiveness of Security Code Review". In: *Engineering Secure Software and Systems*. Springer Berlin Heidelberg, 2013, pp. 197–212.

[78] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel. "An empirical study of cryptographic misuse in android applications". In: *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*. CCS '13. Association for Computing Machinery, 2013, pp. 73–84.

[79] Y. Elrakaiby, M. Amrani, and Y. Le Traon. “Security@Runtime: A Flexible MDE Approach to Enforce Fine-grained Security Policies”. In: *Engineering Secure Software and Systems*. Springer International Publishing, 2014, pp. 19–34.

[80] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. “TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones”. In: *ACM Trans. Comput. Syst.* 32.2 (2014).

[81] J. Epstein, S. Matsumoto, and G. McGraw. “Software security and SOA: danger, Will Robinson!” In: *IEEE Security & Privacy* 4.1 (2006), pp. 80–83.

[82] S. Ereth, H. Mantel, and M. Perner. *Towards a Common Specification Language for Information-Flow Security in Rs 3 and beyond: Rifl 1.0–the Language*. Technical Report TUD-CS-2014-0115, TU Darmstadt, 2014.

[83] M. Esposito, M. Robredo, F. Arcelli Fontana, and V. Lenarduzzi. “On the correlation between architectural smells and static analysis warnings”. In: *Software Quality Journal* 33.4 (2025), p. 33.

[84] S. Fahl, M. Harbach, T. Muders, L. Baumgärtner, B. Freisleben, and M. Smith. “Why eve and mallory love android: an analysis of android SSL (in)security”. In: *Proceedings of the 2012 ACM Conference on Computer and Communications Security*. CCS ’12. Association for Computing Machinery, 2012, pp. 50–61.

[85] S. Fakhoury, A. Naik, G. Sakkas, S. Chakraborty, and S. K. Lahiri. “LLM-based Test-Driven Interactive Code Generation: User Study and Empirical Evaluation”. In: *IEEE Transactions on Software Engineering* 50.9 (2024), pp. 2254–2268.

[86] M. Fang and M. Hafiz. “Discovering buffer overflow vulnerabilities in the wild: an empirical study”. In: *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. ESEM ’14. Association for Computing Machinery, 2014.

[87] Federal Office for Information Security (BSI). *BSI TR-02102 Cryptographic Mechanisms*. (Visited on 01/23/2026).

[88] Federal Trade Commission. *Equifax Data Breach Settlement. Federal Trade Commission*. 11, 2019.

[89] E. Fernández-Medina, J. Trujillo, R. Villarroel, and M. Piattini. “Developing secure data warehouses with a UML extension”. In: *Information Systems* 32.6 (2007), pp. 826–856.

[90] P. Ferrara, L. Olivieri, and F. Spoto. “Tailoring Taint Analysis to GDPR”. In: *Privacy Technologies and Policy*. Springer International Publishing, 2018, pp. 63–76.

[91] T. Fink, M. Koch, and K. Pauls. “An MDA approach to Access Control Specifications Using MOF and UML Profiles”. In: *Electronic Notes in Theoretical Computer Science* 142 (2006). Proceedings of the First International Workshop on Views on Designing Complex Architectures (VODCA 2004), pp. 161–179.

[92] F. A. Fontana, I. Pigazzini, R. Roveda, D. Tamburri, M. Zanoni, and E. Di Nitto. “Arcan: A Tool for Architectural Smells Detection”. In: *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*. 2017, pp. 282–285.

[93] J. Gajrani, M. Tripathi, V. Laxmi, G. Somani, A. Zemmari, and M. S. Gaur. "Vulvet: Vetting of Vulnerabilities in Android Apps to Thwart Exploitation". In: *Digital Threats* 1.2 (2020).

[94] G. Georg, K. Anastasakis, B. Bordbar, S. H. Houmb, I. Ray, and M. Toahchoodee. "Verification and Trade-Off Analysis of Security Properties in UML System Models". In: *IEEE Transactions on Software Engineering* 36.3 (2010), pp. 338–356.

[95] C. Gerking, W. Schäfer, S. Dziwok, and C. Heinzemann. "Domain-Specific Model Checking for Cyber-Physical Systems." In: *MoDeVVa@ models*. 2015, pp. 18–27.

[96] C. Gerking and D. Schubert. "Component-Based Refinement and Verification of Information-Flow Security Policies for Cyber-Physical Microservice Architectures". In: *2019 IEEE International Conference on Software Architecture (ICSA)*. 2019, pp. 61–70.

[97] C. Gerking, D. Schubert, and E. Bodden. "Model Checking the Information Flow Security of Real-Time Systems". In: *Engineering Secure Software and Systems*. Springer International Publishing, 2018, pp. 27–43.

[98] D. Giffhorn and C. Hammer. "Precise Analysis of Java Programs Using JOANA". In: *2008 Eighth IEEE International Working Conference on Source Code Analysis and Manipulation*. 2008, pp. 267–268.

[99] M. Giordano, G. Polese, G. Scanniello, and G. Tortora. "A system for visual role-based policy modelling". In: *Journal of Visual Languages & Computing* 21.1 (2010), pp. 41–64.

[100] GitHub. *CodeQL Website*. 2024. URL: `https://codeql.github.com/` (visited on 02/09/2026).

[101] H. Gomaa and M. Eonsuk Shin. "Modelling complex systems by separating application and security concerns". In: *Proceedings. Ninth IEEE International Conference on Engineering of Complex Computer Systems*. 2004, pp. 19–28.

[102] Google. *ErrorProne Website*. 2024.

[103] K. Goseva-Popstojanova and A. Perhinschi. "On the capability of static code analysis to detect security vulnerabilities". In: *Information and Software Technology* 68 (2015), pp. 18–33.

[104] S. Greiner and M. Herda. *CoCoME with Security*. Tech. rep. 2. Karlsruher Institut für Technologie (KIT), 2017. 30 pp.

[105] S. Greiner, M. Mohr, and B. Beckert. "Modular Verification of Information Flow Security in Component-Based Systems". In: *Software Engineering and Formal Methods*. Springer International Publishing, 2017, pp. 300–315.

[106] F. Greis. *Elektronische Patientenakte: So lässt sich auf die ePAs aller Versicherten zugreifen*. 2024. URL: `https://www.golem.de/news/elektronische-patientenakte-so-laesst-sich-auf-die-epa-aller-versicherten-zugreifen-2412-192003.html` (visited on 02/09/2026).

[107] L. A. Gunawan, P. Herrmann, and F. A. Kraemer. “Towards the Integration of Security Aspects into System Development Using Collaboration-Oriented Models”. In: *Security Technology*. Springer Berlin Heidelberg, 2009, pp. 72–85.

[108] L. A. Gunawan, F. A. Kraemer, and P. Herrmann. “A Tool-Supported Method for the Design and Implementation of Secure Distributed Applications”. In: *Engineering Secure Software and Systems*. Springer Berlin Heidelberg, 2011, pp. 142–155.

[109] F. Gutierrez. *Introducing Spring Framework*. Apress, 2014.

[110] K. L. Gwet. “Computing Inter-Rater Reliability and Its Variance in the Presence of High Agreement”. In: *British Journal of Mathematical and Statistical Psychology* 61.1 (2008), pp. 29–48.

[111] S. Hahner. “Architecture-Based and Uncertainty-Aware Confidentiality Analysis”. PhD thesis. Karlsruher Institut für Technologie (KIT), 2025. 296 pp.

[112] S. Hahner, T. Bitschi, M. Walter, T. Bureš, P. Hnětynka, and R. Heinrich. “Model-based Confidentiality Analysis under Uncertainty”. In: *2023 IEEE 20th International Conference on Software Architecture Companion (ICSA-C)*. 2023, pp. 256–263.

[113] S. Hahner, S. Seifermann, R. Heinrich, M. Walter, T. Bureš, and P. Hnětynka. “Modeling Data Flow Constraints for Design-Time Confidentiality Analyses”. In: *2021 IEEE 18th International Conference on Software Architecture Companion (ICSA-C)*. 2021, pp. 15–21.

[114] J. Haltermann and H. Wehrheim. “Exchanging Information in Cooperative Software Validation”. In: *Software and Systems Modeling* 23.3 (2024), pp. 695–719.

[115] D. Harel and B. Rumpe. “Meaningful modeling: what’s the semantics of "semantics"?” In: *Computer* 37.10 (2004), pp. 64–72.

[116] J. Häring. *Enabling the Information Transfer between Architecture and Source Code for Security Analysis*. Abschlussarbeit - Bachelor. Karlsruher Institut für Technologie (KIT) / Karlsruher Institut für Technologie (KIT), 2021.

[117] M. Hazhirpasand, M. Ghafari, and O. Nierstrasz. “Java Cryptography Uses in the Wild”. In: *Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. ESEM ’20. Association for Computing Machinery, 2020.

[118] Á. Hegedüs, G. Bergmann, I. Ráth, and D. Varró. “Back-annotation of Simulation Traces with Change-Driven Model Transformations”. In: *2010 8th IEEE International Conference on Software Engineering and Formal Methods*. 2010, pp. 145–155.

[119] F. Heidenreich, J. Johannes, M. Seifert, and C. Wende. “Closing the Gap between Modelling and Java”. In: *Software Language Engineering*. Springer Berlin Heidelberg, 2010, pp. 374–383.

[120] R. Heinrich. “Architectural runtime models for integrating runtime observations and component-based models”. In: *Journal of Systems and Software* 169 (2020), p. 110722.

[121] R. Heinrich. “Architecture-based Evolution of Dependable Software-intensive Systems”. Habilitation. Karlsruher Institut für Technologie (KIT), 2023. 126 pp.

[122] *Composing Model-Based Analysis Tools*. 1st ed. Springer International Publishing, 2021.

[123] R. Heinrich, M. Strittmatter, and R. Reussner. “A Layered Reference Architecture for Metamodels to Tailor Quality Modeling and Analysis”. In: *IEEE Transactions on Software Engineering* 47.4 (2021), pp. 775–800.

[124] R. Heldal and F. Hultin. “Bridging Model-Based and Language-Based Security”. In: *Computer Security – ESORICS 2003*. Springer Berlin Heidelberg, 2003, pp. 235–252.

[125] D. Helm, F. Kübler, M. Reif, M. Eichberg, and M. Mezini. “Modular collaborative program analysis in OPAL”. In: *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2020. Association for Computing Machinery, 2020, pp. 184–196.

[126] K. Hermann, S. Peldszus, J.-P. Steghöfer, and T. Berger. “An Exploratory Study on the Engineering of Security Features”. In: *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. 2025, pp. 2470–2482.

[127] S. Herold, H. Klus, Y. Welsch, C. Deiters, A. Rausch, R. Reussner, K. Krogmann, H. Koziolek, R. Mirandola, B. Hummel, M. Meisinger, and C. Pfaller. “CoCoME - The Common Component Modeling Example”. In: *The Common Component Modeling Example: Comparing Software Component Models*. Springer Berlin Heidelberg, 2008, pp. 16–53.

[128] A. Herzog, N. Shahmehri, and C. Duma. “An Ontology of Information Security”. In: *International Journal of Information Security and Privacy* 1.4 (2007), pp. 1–23.

[129] T. Hettel, M. Lawley, and K. Raymond. “Model Synchronisation: Definitions for Round-Trip Engineering”. In: *Theory and Practice of Model Transformations*. Springer Berlin Heidelberg, 2008, pp. 31–45.

[130] S. Hidaka, M. Tisi, J. Cabot, and Z. Hu. “Feature-based classification of bidirectional transformation approaches”. In: *Software & Systems Modeling* 15.3 (2016), pp. 907–928.

[131] R. Hofer and M. Loper. “DIS Today [Distributed Interactive Simulation]”. In: *Proceedings of the IEEE* 83.8 (1995), pp. 1124–1137.

[132] B. Hoisl, S. Sobernig, and M. Strembeck. “Modeling and enforcing secure object flows in process-driven SOAs: an integrated model-driven approach”. In: *Software & Systems Modeling* 13.2 (2014), pp. 513–548.

[133] J.-M. Horcas, M. Pinto, and L. Fuentes. “An automatic process for weaving functional quality attributes using a software product line approach”. In: *Journal of Systems and Software* 112 (2016), pp. 78–95.

[134] J.-M. Horcas, M. Pinto, L. Fuentes, W. Mallouli, and E. Montes de Oca. “An approach for deploying and monitoring dynamic security policies”. In: *Computers & Security* 58 (2016), pp. 20–38.

[135] K. Hough and J. Bell. “A Practical Approach for Dynamic Taint Tracking with Control-flow Relationships”. In: *ACM Trans. Softw. Eng. Methodol.* 31.2 (2021).

[136] D. Hovemeyer and W. Pugh. “Finding bugs is easy”. In: *SIGPLAN Not.* 39.12 (2004), pp. 92–106.

[137] Y.-C. Hu, A. Perrig, and D. Johnson. “Packet leashes: a defense against wormhole attacks in wireless networks”. In: *IEEE INFOCOM 2003. Twenty-second Annual Joint Conference of the IEEE Computer and Communications Societies (IEEE Cat. No.03CH37428)*. Vol. 3. 2003, 1976–1986 vol.3.

[138] N. Imtiaz, S. Thorn, and L. Williams. “A comparative study of vulnerability reporting by software composition analysis tools”. In: *Proceedings of the 15th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. ESEM ’21. Association for Computing Machinery, 2021.

[139] ISO/IEC. *Information Security, Cybersecurity and Privacy Protection — Information Security Controls*. Standard ISO/IEC 27000:2018. International Organization for Standardization, 2018.

[140] ISO/IEC. *Information Security, Cybersecurity and Privacy Protection — Information Security Controls*. Standard ISO/IEC 27001:2022. International Organization for Standardization, 2022.

[141] ISO/IEC. *Software Product Quality Requirements and Evaluation (SQuaRE) – Product Quality Model*. Standard 25010:2023. International Organization for Standardization (ISO), 2023.

[142] ISO/IEC/IEEE. *Systems and Software Engineering — Architecture Description*. Standard ISO/IEC/IEEE 42010:2022(E). International Organization for Standardization, 2022.

[143] H. Jakob, N. Loriant, and C. Consel. “An aspect-oriented approach to securing distributed systems”. In: *Proceedings of the 2009 International Conference on Pervasive Services*. ICPS ’09. Association for Computing Machinery, 2009, pp. 21–30.

[144] P. Jansen. *TIOBE Index for January 2025*. 2025. URL: `https://www.tiobe.com/tiobe-index/` (visited on 02/09/2026).

[145] S. Jasser. “Enforcing Architectural Security Decisions”. In: *2020 IEEE International Conference on Software Architecture (ICSA)*. 2020, pp. 35–45.

[146] A. M. Jimenez and S. J. Zepeda. “A Comparison of Gwet’s AC1 and Kappa When Calculating Inter-Rater Reliability Coefficients in a Teacher Evaluation Context”. In: *Journal of Education Human Resources* 38.3 (2020), pp. 290–300.

[147] Á. Jiménez, J. M. Vara, V. A. Bollati, and E. Marcos. “MeTAGeM-Trace: Improving trace generation in model transformation by leveraging the role of transformation models”. In: *Science of Computer Programming* 98 (2015). Fifth issue of Experimental Software and Toolkits (EST): A special issue on Academics Modelling with Eclipse (ACME2012), pp. 3–27.

[148] W. Jin, S. Ullah, D. Yoo, and H. Oh. “NPDHunter: Efficient Null Pointer Dereference Vulnerability Detection in Binary”. In: *IEEE Access* 9 (2021), pp. 90153–90169.

[149] S. Joel, J. Wu, and F. Fard. "A Survey on LLM-based Code Generation for Low-Resource and Domain-Specific Programming Languages". In: *ACM Trans. Softw. Eng. Methodol.* (2025). Just Accepted.

[150] Johannes Geismann and Bastian Haverkamp and Eric Bodden. "Ensuring Threat-Model Assumptions by Using Static Code Analyses". In: *ECSA 2021 Companion Volume*. CEUR-WS.org.

[151] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge. "Why don't software developers use static analysis tools to find bugs?" In: *2013 35th International Conference on Software Engineering (ICSE)*. 2013, pp. 672–681.

[152] N. P. Johnson, J. Fix, S. R. Beard, T. Oh, T. B. Jablin, and D. I. August. "A collaborative dependence analysis framework". In: *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2017, pp. 148–159.

[153] Joint Task Force Transformation Initiative. *Guide for ConductingRisk Assessments*. Special Publication SP 800-30. National Institute of Standards and Technology, 2012.

[154] J. Jürjens. *Secure Systems Development with UML*. Springer, 2005.

[155] J. Jürjens, J. Schreck, and P. Bartmann. "Model-based security analysis for mobile communications". In: *Proceedings of the 30th International Conference on Software Engineering*. ICSE '08. Association for Computing Machinery, 2008, pp. 683–692.

[156] A. Kaddani, A. Baina, and L. Echabbi. "Towards a Model Driven Security for critical infrastructures using OrBAC". In: *2014 International Conference on Multimedia Computing and Systems (ICMCS)*. 2014, pp. 1235–1240.

[157] A. Kaplan, T. Kühn, S. Hahner, N. Benkler, J. Keim, D. Fuchss, S. Corallo, and R. Heinrich. "Introducing an Evaluation Method for Taxonomies". In: EASE '22. Association for Computing Machinery, 2022, pp. 311–316.

[158] K. Katkalov, P. Fischer, K. Stenzel, N. Moebius, and W. Reif. "Evaluation of Jif and Joana as Information Flow Analyzers in a Model-Driven Approach". In: *Data Privacy Management and Autonomous Spontaneous Security*. Springer Berlin Heidelberg, 2013, pp. 174–186.

[159] K. Katkalov, K. Stenzel, M. Borek, and W. Reif. "Model-Driven Development of Information Flow-Secure Systems with IFlow". In: *2013 International Conference on Social Computing*. 2013, pp. 51–56.

[160] M. Kellogg, M. Schäf, S. Tasiran, and M. D. Ernst. "Continuous Compliance". In: *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. ASE '20: 35th IEEE/ACM International Conference on Automated Software Engineering. ACM, 21, 2020, pp. 511–523.

[161] M. Kern, F. Erata, M. Iser, C. Sinz, F. Loiret, S. Otten, and E. Sax. "Integrating Static Code Analysis Toolchains". In: *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*. Vol. 1. 2019, pp. 523–528.

[162] D.-k. Kim and P. Gokhale. "A Pattern-Based Technique for Developing UML Models of Access Control Systems". In: *30th Annual International Computer Software and Applications Conference (COMPSAC'06)*. Vol. 1. 2006, pp. 317–324.

[163] S. Kim, D.-K. Kim, L. Lu, S. Kim, and S. Park. "A feature-based approach for modeling role-based access control systems". In: *Journal of Systems and Software* 84.12 (2011), pp. 2035–2052.

[164] Y. R. Kirschner, M. Gstür, T. Sağlam, S. Weber, and A. Koziolek. "Retriever: A View-Based Approach to Reverse Engineering Software Architecture Models". In: *Journal of Systems and Software* 220 (2025), p. 112277.

[165] Y. R. Kirschner, M. Walter, F. Bossert, R. Heinrich, and A. Koziolek. "Automatic Derivation of Vulnerability Models for Software Architectures". In: *2023 IEEE 20th International Conference on Software Architecture Companion (ICSA-C)*. 2023, pp. 276–283.

[166] B. Kitchenham, S. Pfleeger, L. Pickard, P. Jones, D. Hoaglin, K. El Emam, and J. Rosenberg. "Preliminary Guidelines for Empirical Research in Software Engineering". In: *IEEE Transactions on Software Engineering* 28.8 (2002), pp. 721–734.

[167] H. Klare, M. E. Kramer, M. Langhammer, D. Werle, E. Burger, and R. Reussner. "Enabling consistency in view-based system development - The Vitruvius approach". In: *Journal of Systems and Software* 171 (2021), p. 110815.

[168] M. Koch, L. V. Mancini, and F. Parisi-Presicce. "A graph-based formalism for RBAC". In: *ACM Trans. Inf. Syst. Secur.* 5.3 (2002), pp. 332–365.

[169] S. G. Koch. "A Reference Structure for Modular Model-Based Analyses". PhD thesis. Karlsruher Institut für Technologie (KIT) / Karlsruher Institut für Technologie (KIT), 2023. 276 pp.

[170] M. Konersmann, A. Kaplan, T. Kühn, R. Heinrich, A. Koziolek, R. Reussner, J. Jürjens, M. al-Doori, N. Boltz, M. Ehl, D. Fuchß, K. Großer, S. Hahner, J. Keim, M. Lohr, T. Sağlam, S. Schulz, and J.-P. Töberg. "Evaluation Methods and Replicability of Software Architecture Research Objects". In: *2022 IEEE 19th International Conference on Software Architecture (ICSA)*. 2022, pp. 157–168.

[171] M. Konersmann, B. Rumpe, M. Stachon, S. Stüber, and V. Voufo. "Towards a Semantically Useful Definition of Conformance with a Reference Model." In: *The Journal of Object Technology* 23.3 (2024), p. 1.

[172] M. E. Kramer, M. Hecker, S. Greiner, K. Bao, and K. Yurchenko. *Model-Driven Specification and Analysis of Confidentiality in Component-Based Systems*. 2017.

[173] M. E. Kramer, M. Langhammer, D. Messinger, S. Seifermann, and E. Burger. "Change-Driven Consistency for Component Code, Architectural Models, and Contracts". In: *Proceedings of the 18th International ACM SIGSOFT Symposium on Component-Based Software Engineering*. CBSE '15. Association for Computing Machinery, 2015, pp. 21–26.

[174] M. Krausz, S. Peldszus, F. Regazzoni, T. Berger, and T. Güneysu. *120 Domain-Specific Languages for Security*. 2024. arXiv: `2408.06219 [cs.CR]`.

[175] K. Krippendorff. *Content Analysis: An Introduction to Its Methodology*. SAGE Publications, Inc., 2019.

[176] S. Krüger, S. Nadi, M. Reif, K. Ali, M. Mezini, E. Bodden, F. Göpfert, F. Günther, C. Weinert, D. Demmler, and R. Kamath. "CogniCrypt: Supporting developers in using cryptography". In: *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2017, pp. 931–936.

[177] T. Kuhr, T. Forster, T. Braun, and R. Gotzhein. "FERAL — Framework for simulator coupling on requirements and architecture level". In: *2013 Eleventh ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2013)*. 2013, pp. 11–22.

[178] M. Kulenovic and D. Donko. "A survey of static code analysis methods for security vulnerabilities detection". In: *2014 37th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. 2014, pp. 1381–1386.

[179] R. Kumar. "A Model-Based Safety-Security Risk Analysis Framework for Interconnected Critical Infrastructures". In: *Critical Infrastructure Protection XIV*. Springer International Publishing, 2020, pp. 283–306.

[180] R. Kumar and M. Stoelinga. "Quantitative Security and Safety Analysis with Attack-Fault Trees". In: *2017 IEEE 18th International Symposium on High Assurance Systems Engineering (HASE)*. 2017, pp. 25–32.

[181] M. Langhammer, S. Lehrig, and M. E. Kramer. "Reuse and configuration for code generating architectural refinement transformations". In: *Proceedings of the 1st Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling*. VAO '13. Association for Computing Machinery, 2013.

[182] F. Lanzinger, C. Martin, F. Reiche, S. Teuber, R. Heinrich, and A. Weigl. "Quantifying Software Correctness by Combining Architecture Modeling and Formal Program Analysis". In: *Proceedings of the 39th ACM/SIGAPP Symposium on Applied Computing*. SAC '24. Association for Computing Machinery, 2024, pp. 1702–1711.

[183] F. Lau, S. Rubin, M. Smith, and L. Trajkovic. "Distributed denial of service attacks". In: *Smc 2000 conference proceedings. 2000 ieee international conference on systems, man and cybernetics. 'cybernetics evolving to systems, humans, organizations, and their complex interactions' (cat. no.0*. Vol. 3. 2000, 2275–2280 vol.3.

[184] J. Lawall and G. Muller. "Coccinelle: 10 Years of Automated Evolution in the Linux Kernel". In: *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, 2018, pp. 601–614.

[185] J. Lehmann. "Iterative quelltextanalyse für informationsflusssicherheit zur überprüfung von vertraulichkeit auf architekturebene". MA thesis. Karlsruher Institut für Technologie (KIT) / Karlsruher Institut für Technologie (KIT), 2024. 100 pp.

[186] M. Lehto. "Cyber-Attacks Against Critical Infrastructure". In: *Cyber Security: Critical Infrastructure Protection*. Springer International Publishing, 2022, pp. 3–42.

[187] S. Lerner, D. Grove, and C. Chambers. "Composing Dataflow Analyses and Transformations". In: *SIGPLAN Not.* 37.1 (1, 2002), pp. 270–282.

[188] R. Li, P. Liang, M. Soliman, and P. Avgeriou. "Understanding Architecture Erosion: The Practitioners' Perceptive". In: *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*. 2021, pp. 311–322.

[189] R. Li, P. Liang, M. Soliman, and P. Avgeriou. "Understanding software architecture erosion: A systematic mapping study". In: *Journal of Software: Evolution and Process* 34.3 (2022), e2423.

[190] S. Lipp, S. Banescu, and A. Pretschner. "An empirical study on the effectiveness of static C code analyzers for vulnerability detection". In: *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2022. Association for Computing Machinery, 2022, pp. 544–555.

[191] H. Liu, S. Chen, R. Feng, C. Liu, K. Li, Z. Xu, L. Nie, Y. Liu, and Y. Chen. "A Comprehensive Study on Quality Assurance Tools for Java". In: *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2023. Association for Computing Machinery, 2023, pp. 285–297.

[192] T. Lodderstedt, D. Basin, and J. Doser. "SecureUML: A UML-Based Modeling Language for Model-Driven Security". In: *UML 2002 — The Unified Modeling Language*. Springer Berlin Heidelberg, 2002, pp. 426–441.

[193] M. Lombard, J. Snyder-Duch, and C. C. Bracken. "Content Analysis in Mass Communication: Assessment and Reporting of Intercoder Reliability". In: *Human Communication Research* 28.4 (2006), pp. 587–604. eprint: `https://academic.oup.com/hcr/article-pdf/28/4/587/22337849/jhumcom0587.pdf`.

[194] A. M. Binder and I. Kunz. "Using Static Code Analysis for GDPR Compliance Checks". In: *Computer Security. ESORICS 2024 International Workshops*. Springer Nature Switzerland, 2025, pp. 153–169.

[195] S. Ma, D. Lo, T. Li, and R. H. Deng. "CDRep: Automatic Repair of Cryptographic Misuses in Android Applications". In: *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*. ASIA CCS '16. Association for Computing Machinery, 2016, pp. 711–722.

[196] S. Ma, F. Thung, D. Lo, C. Sun, and R. H. Deng. "VuRLE: Automatic Vulnerability Detection and Repair by Learning from Examples". In: *Computer Security – ESORICS 2017*. Springer International Publishing, 2017, pp. 229–246.

[197] A. Machiry, C. Spensky, J. Corina, N. Stephens, C. Kruegel, and G. Vigna. "DR. CHECKER: A Soundy Analysis for Linux Kernel Drivers". In: *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, 2017, pp. 1007–1024.

[198] H. Mantel. "Information Flow and Noninterference". In: *Encyclopedia of Cryptography and Security*. Springer US, 2011, pp. 605–607.

[199] A. Marback, H. Do, K. He, S. Kondamarri, and D. Xu. "A Threat Model-Based Approach to Security Testing". In: *Software: Practice and Experience* 43.2 (2013), pp. 241–258.

[200] L. Marchezan, W. K. G. Assunção, E. Herac, F. Keplinger, A. Egyed, and C. Lauwerys. "Fulfilling Industrial Needs for Consistency Among Engineering Artifacts". In: *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. 2023, pp. 246–257.

[201] D. Marjamäki et al. *CppCheck Feature Overview*. 2025. URL: `https://cppcheck.sourceforge.io/` (visited on 02/09/2026).

[202] A. Mashkoor, A. Egyed, R. Wille, and S. Stock. "Model-driven engineering of safety and security software systems: A systematic mapping study and future research directions". In: *Journal of Software: Evolution and Process* 35.7 (2022).

[203] M. Mazurek. "We Are the Experts, and We Are the Problem: The Security Advice Fiasco". In: *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. CCS '22. Association for Computing Machinery, 2022, p. 7.

[204] G. McGraw. "Software security". In: *IEEE Security & Privacy* 2.2 (2004), pp. 80–83.

[205] M. Menzel and C. Meinel. "A Security Meta-model for Service-Oriented Architectures". In: *2009 IEEE International Conference on Services Computing*. 2009, pp. 251–259.

[206] M. Menzel, R. Warschofsky, and C. Meinel. "A Pattern-Driven Generation of Security Policies for Service-Oriented Architectures". In: *2010 IEEE International Conference on Web Services*. 2010, pp. 243–250.

[207] B. Merkle. "Stop the Software Architecture Erosion: Building Better Software Systems". In: *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*. Oopsla '10. Association for Computing Machinery, 2010, pp. 129–138.

[208] Meta Platforms. *Pyre/Pysa Website*. 2025. URL: `https://pyre-check.org/docs/pysa-basics/` (visited on 02/09/2026).

[209] B. Meyers, H. Vangheluwe, J. Denil, and R. Salay. "A Framework for Temporal Verification Support in Domain-Specific Modelling". In: *IEEE Transactions on Software Engineering* 46.4 (2020), pp. 362–404.

[210] Microsoft Corporation. *Microsoft Security Development Lifecycle*. URL: `https://www.microsoft.com/en-us/securityengineering/sdl` (visited on 02/09/2026).

[211] *Modeling the Travel Planner Application with IFlow*. URL: `https://kiv.isse.de/projects/iflow/TravelPlannerSite/index.html` (visited on 02/09/2026).

[212] N. Moebius, K. Stenzel, H. Grandy, and W. Reif. "SecureMDD: A Model-Driven Development Method for Secure Smart Card Applications". In: *2009 International Conference on Availability, Reliability and Security*. 2009, pp. 841–846.

[213] D. Monschein, M. Mazkatli, R. Heinrich, and A. Koziolek. "Enabling Consistency between Software Artefacts for Software Adaption and Evolution". In: *2021 IEEE 18th International Conference on Software Architecture (ICSA)*. 2021, pp. 1–12.

[214] O. d. Moor, M. Verbaere, E. Hajiyev, P. Avgustinov, T. Ekman, N. Ongkingco, D. Sereni, and J. Tibble. "Keynote Address: .QL for Source Code Analysis". In: *Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007)*. 2007, pp. 3–16.

[215] S. Moral-García, S. Moral-Rubio, E. B. Fernández, and E. Fernández-Medina. "Enterprise security pattern: A model-driven architecture instance". In: *Comput. Stand. Interfaces* 36.4 (2014), pp. 748–758.

[216] B. Morin, T. Mouelhi, F. Fleurey, Y. Le Traon, O. Barais, and J.-M. Jézéquel. "Security-driven model-based dynamic adaptation". In: *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering*. ASE '10. Association for Computing Machinery, 2010, pp. 205–214.

[217] T. Mouelhi, F. Fleurey, B. Baudry, and Y. Traon. "A Model-Based Framework for Security Policy Specification, Deployment and Testing". In: *Proceedings of the 11th International Conference on Model Driven Engineering Languages and Systems*. MoDELS '08. Springer-Verlag, 2008, pp. 537–552.

[218] D. Mouheb, C. Talhi, A. Mourad, V. Lima, M. Debbabi, L. Wang, and M. Pourzandi. "An Aspect-Oriented Approach for Software Security Hardening: from Design to Implementation". In: *Proceedings of the 2009 Conference on New Trends in Software Methodologies, Tools and Techniques: Proceedings of the Eighth SoMeT 09*. IOS Press, 2009, pp. 203–222.

[219] H. Mouratidis and P. Giorgini. "Secure Tropos: A Security-Oriented Extension of the Tropos Methodology". In: *International Journal of Software Engineering and Knowledge Engineering* 17.02 (2007), pp. 285–309.

[220] A. Mousa, M. Karabatak, and T. Mustafa. "Database Security Threats and Challenges". In: *2020 8th International Symposium on Digital Forensics and Security (ISDFS)*. 2020, pp. 1–5.

[221] D. C. Muñoz Hurtado, S. Serbout, and C. Pautasso. "Mining Security Documentation Practices in OpenAPI Descriptions". In: *2025 IEEE 22nd International Conference on Software Architecture (ICSA)*. 2025, pp. 119–130.

[222] P. Muntean, A. Rabbi, A. Ibing, and C. Eckert. "Automated Detection of Information Flow Vulnerabilities in UML State Charts and C Code". In: *2015 IEEE International Conference on Software Quality, Reliability and Security - Companion*. 2015, pp. 128–137.

[223] I. Muslukhov, Y. Boshmaf, and K. Beznosov. "Source Attribution of Cryptographic API Misuse in Android Applications". In: *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*. ASIACCS '18. Association for Computing Machinery, 2018, pp. 133–146.

[224] A. C. Myers. "JFlow: practical mostly-static information flow control". In: *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '99. Association for Computing Machinery, 1999, pp. 228–241.

[225] M. Nachtigall, M. Schlichtig, and E. Bodden. "A large-scale study of usability criteria addressed by static analysis tools". In: *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2022. Association for Computing Machinery, 2022, pp. 532–543.

[226] S. Nadi, S. Krüger, M. Mezini, and E. Bodden. "Jumping through hoops: why do Java developers struggle with cryptography APIs?" In: *Proceedings of the 38th International Conference on Software Engineering*. ICSE '16. Association for Computing Machinery, 2016, pp. 935–946.

[227] Y. Nakamura, M. Tatsubori, T. Imamura, and K. Ono. "Model-driven security based on a Web services security architecture". In: *2005 IEEE International Conference on Services Computing (SCC'05) Vol-1*. Vol. 1. 2005, 7–15 vol.1.

[228] National Institute of Standards and Technology (NIST). *Source Code Security Analyzers*. 2021. URL: https://www.nist.gov/itl/ssd/software-quality-group/source-code-security-analyzers (visited on 01/15/2026).

[229] R. Neisse and J. Doerr. "Model-based specification and refinement of usage control policies". In: *2013 Eleventh Annual Conference on Privacy, Security and Trust*. 2013, pp. 169–176.

[230] K. L. Newbury. "Automated Hotfixes for Misuses of Crypto Apis". MA thesis. University of Alberta, 2020.

[231] P. H. Nguyen, M. Kramer, J. Klein, and Y. L. Traon. "An extensive systematic review on the Model-Driven Development of secure systems". In: *Information and Software Technology* 68 (2015), pp. 62–81.

[232] P. H. Nguyen, G. Nain, J. Klein, T. Mouelhi, and Y. Le Traon. "Modularity and Dynamic Adaptation of Flexibly Secure Systems: Model-Driven Adaptive Delegation in Access Control Management". In: *Transactions on Aspect-Oriented Software Development XI*. Springer Berlin Heidelberg, 2014, pp. 109–144.

[233] OASIS. *Web Services Business Process Execution Language*.

[234] Object Management Group. *About the Meta Object Facility Specification Version 2.5.1*. URL: https://www.omg.org/spec/MOF (visited on 02/09/2026).

[235] Object Management Group. *Unified Modeling Language, v2.5.1*. URL: https://www.omg.org/spec/UML/2.5.1 (visited on 02/10/2026).

[236] T. Ohyama. "Statistical Inference of Gwet's AC1 Coefficient for Multiple Raters and Binary Outcomes". In: *Communications in Statistics - Theory and Methods* 50.15 (2020), pp. 3564–3572.

[237] E. Oladimeji, L. Chung, and S. Supakkul. "A Model-Driven Approach to Architecting Secure Software". In: *International Conference on Software Engineering & Knowledge Engineering*. 2007.

[238] OMG. *OMG Document – Ormsc/10-09-06 (MDA Foundation Model - Cambridge MA AB Edits Draft, Clean)*.

[239] OWASP Foundation. *A02 Cryptographic Failures - OWASP Top*. URL: https://owasp.org/Top10/2025/A04_2025-Cryptographic_Failures/ (visited on 02/09/2026).

[240] OWASP Foundation. *OWASP Dependency-Check*. 2022. URL: https://owasp.org/www-project-dependency-check/ (visited on 02/09/2026).

[241] OWASP Foundation. *OWASP Source Code Analysis Tools List*.

[242] OWASP Foundation. *OWASP Top Ten | OWASP Foundation*. URL: https://owasp.org/Top10/2025/ (visited on 02/09/2026).

[243] R. Paletov, P. Tsankov, V. Raychev, and M. Vechev. "Inferring crypto API rules from code changes". In: *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2018. Association for Computing Machinery, 2018, pp. 450–464.

[244] L. Passos, R. Terra, M. T. Valente, R. Diniz, and N. Mendonça. "Static Architecture-Conformance Checking: An Illustrative Overview". In: *IEEE Software* 27.5 (2010), pp. 82–89.

[245] N. W. Paton and O. Díaz. "Active database systems". In: *ACM Computing Surveys* 31.1 (1999), pp. 63–103.

[246] F. Pauck and H. Wehrheim. "Together strong: cooperative Android app analysis". In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2019. Association for Computing Machinery, 2019, pp. 374–384.

[247] J. A. Pavlich-Mariscal, S. A. Demurjian, and L. D. Michel. "A framework of composable access control features: Preserving separation of access control concerns from models to code". In: *Computers & Security* 29.3 (2010). Special issue on software engineering for secure systems, pp. 350–379.

[248] M. V. Pawar and J. Anuradha. "Network Security and Types of Attacks in Network". In: *Procedia Computer Science* 48 (2015). International Conference on Computer, Communication and Convergence (ICCC 2015), pp. 503–506.

[249] R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, and L. Seinturier. "Spoon: A Library for Implementing Analyses and Transformations of Java Source Code". In: *Software: Practice and Experience* 46 (2015), pp. 1155–1179.

[250] S. Peldszus. *Security Compliance in Model-Driven Development of Software Systems in Presence of Long-Term Evolution and Variants*. Springer, 2022.

[251] S. Peldszus, J. Bürger, and J. Jürjens. "UMLsecRT: Reactive Security Monitoring of Java Applications With Round-Trip Engineering". In: *IEEE Transactions on Software Engineering* 50.1 (2024), pp. 16–47.

[252] S. Peldszus, J. Bürger, T. Kehrer, and J. Jürjens. "Ontology-driven evolution of software security". In: *Data & Knowledge Engineering* 134 (2021), p. 101907.

[253] S. Peldszus, K. Großer, M. Konersmann, W. Brunotte, M. Ahrens, K. Schneider, and J. Jürjens. "Too Many Issues: Automatically Prioritizing Analyzer Findings by Tracing Security Importance". In: *ACM Trans. Softw. Eng. Methodol.* (2025). Just Accepted.

[254] S. Peldszus, F. Reiche, K. Hermann, S. Corallo, T. Berger, and R. Heinrich. *Can I Check What I Designed? Mapping Security Design DSLs to Code Analyzers*. 2026. arXiv: 2605.07814 [cs.CR].

[255] S. Peldszus, D. Strüber, and J. Jürjens. "Model-based security analysis of feature-oriented software product lines". In: *SIGPLAN Not.* 53.9 (2020), pp. 93–106.

[256] S. Peldszus, K. Tuma, D. Strüber, J. Jürjens, and R. Scandariato. "Secure Data-Flow Compliance Checks between Models and Code Based on Automated Mappings". In: *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS)*. 2019, pp. 23–33.

[257] S. M. Peldszus. "The GRaViTY Framework". In: *Security Compliance in Model-Driven Development of Software Systems in Presence of Long-Term Evolution and Variants*. Springer, 2022, pp. 383–392.

[258] T. Pereira and H. Santos. "An Ontology Approach in Designing Security Information Systems to Support Organizational Security Risk Knowledge". In: *Proceedings of the International Conference on Knowledge Engineering and Ontology Development (IC3K 2012) - Volume 1: SSEO*. INSTICC. SciTePress, 2012, pp. 461–466.

[259] M. Pinto, N. Gámez, L. Fuentes, M. Amor, J. M. Horcas, and I. Ayala. "Dynamic Reconfiguration of Security Policies in Wireless Sensor Networks". In: *Sensors* 15.3 (2015), pp. 5251–5280.

[260] PMD Open Source Project. *PMD*. 2024. URL: https://docs.pmd-code.org/latest/index.html (visited on 02/09/2026).

[261] N. Polikarpova, C. A. Furia, Y. Pei, Y. Wei, and B. Meyer. "What good are strong specifications?" In: *2013 35th International Conference on Software Engineering (ICSE)*. 2013, pp. 262–271.

[262] C. Ptolemaeus. *System Design, Modeling, and Simulation: Using Ptolemy II*. Vol. 1. Ptolemy. org Berkeley, 2014.

[263] PyCQA. *Bandit Website*. 2025. URL: https://github.com/PyCQA/bandit (visited on 02/09/2026).

[264] W. Rafnsson, R. Giustolisi, M. Kragerup, and M. Høyrup. "Fixing Vulnerabilities Automatically with Linters". In: *Network and System Security*. Springer International Publishing, 2020, pp. 224–244.

[265] S. Rahaman, Y. Xiao, S. Afrose, F. Shaon, K. Tian, M. Frantz, M. Kantarcioglu, and D. (Yao. "CryptoGuard: High Precision Detection of Cryptographic Vulnerabilities in Massive-sized Java Projects". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. CCS '19. Association for Computing Machinery, 2019, pp. 2455–2472.

[266] P. Ralph et al. *SIGSOFT Empirical Standards*. 2023.

[267] I. Ray, R. France, N. Li, and G. Georg. "An aspect-based approach to modeling access control concerns". In: *Information and Software Technology* 46.9 (2004), pp. 575–587.

[268] F. Reiche et al. *Architecture-And-StaticCode-Analyses-Coupling-Framework*.

[269] F. Reiche et al. *Pattern-Search-Based-Coupling*. URL: https://https://github.com/CASCADE - AnalysisCouplingCouplingPatternSearchBasedSourceCodeAnalyses (visited on 02/09/2026).

[270] F. Reiche. *Replication Package For The Thesis Coupling Architectural Analyses and Static Source Code Analyses for Detecting Security Vulnerabilities*. Zenodo, 2026.

[271] F. Reiche and R. Heinrich. "Detecting Encryption Vulnerabilities by Coupling Architectural Analyses and Source Code Analyses". In: *2025 IEEE 22nd International Conference on Software Architecture Companion (ICSA-C)*. 2025, pp. 370–379.

[272] F. Reiche, R. Reussner, and R. Heinrich. "Detecting Information Flow Security Vulnerabilities by Analysis Coupling". In: *IEEE Transactions on Software Engineering* 51.10 (2025), pp. 2710–2743.

[273] F. Reiche, T. Weber, S. Becker, S. Weber, R. Heinrich, and E. Burger. "Consistency Management for Security Annotations for Continuous Verification". In: *Proceedings of the ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems*. MODELS Companion '24. Association for Computing Machinery, 2024, pp. 1096–1105.

[274] J. Ren and R. N. Taylor. "A Secure Software Architecture Description Language". In: *Workshop on Software Security Assurance Tools, Techniques, and Metrics*. 2005.

[275] R. H. Reussner, S. Becker, J. Happe, R. Heinrich, A. Koziolek, H. Koziolek, M. Kramer, and K. Krogmann. *Modeling and Simulating Software Architectures - the Palladio Approach*. MIT Press, 2016. 408 pp.

[276] J. Reznik, T. Ritter, R. Schreiner, and U. Lang. "Model Driven Development of Security Aspects". In: *Electronic Notes in Theoretical Computer Science* 163.2 (2007). Proceedings of the Second International Workshop on Aspect-Based and Model-Based Separation of Concerns in Software Systems (ABMB 2006), pp. 65–79.

[277] C. Richter, M. Chalupa, M.-C. Jakobs, and H. Wehrheim. "Cooperative Software Verification via Dynamic Program Splitting". In: *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. 2025, pp. 2087–2099.

[278] A. Robles-González, J. Parra-Arnau, and J. Forné. "A LINDDUN-Based framework for privacy threat analysis on identification and authentication processes". In: *Computers & Security* 94 (2020), p. 101755.

[279] R. Ross, M. Winstead, and M. McEvilley. *Engineering Trustworthy Secure Systems*. Special Publication SP 800-160. National Institute of Standards and Technology, 2022.

[280] T. Roth, J. Näumann, D. Helm, S. Keidel, and M. Mezini. "AXA: Cross-Language Analysis through Integration of Single-Language Analyses". In: *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. ASE '24. Association for Computing Machinery, 2024, pp. 1195–1205.

[281] J. F. Ruiz, M. Arjona, A. Maña, and C. Rudolph. "Security knowledge representation artifacts for creating secure IT systems". In: *Computers & Security* 64 (2017), pp. 69–91.

[282] P. Runeson and M. Höst. “Guidelines for conducting and reporting case study research in software engineering”. In: *Empirical Software Engineering* 14.2 (2009), pp. 131–164.

[283] T. Runge, A. Kittelmann, M. Servetto, A. Potanin, and I. Schaefer. “Information Flow Control-by-Construction for an Object-Oriented Language”. In: *Software Engineering and Formal Methods*. Springer International Publishing, 2022, pp. 209–226.

[284] I. Ryan, U. Roedig, and K.-J. Stol. “Unhelpful Assumptions in Software Security Research”. In: *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’23. Association for Computing Machinery, 2023, pp. 3460–3474.

[285] A. Sabelfeld and A. Myers. “Language-based information-flow security”. In: *IEEE Journal on Selected Areas in Communications* 21.1 (2003), pp. 5–19.

[286] A. Sadeghi, F. Aminmansour, and H. R. Shahriari. “Tiny jump-oriented programming attack (A class of code reuse attacks)”. In: *2015 12th International Iranian Society of Cryptology Conference on Information Security and Cryptology (ISCISC)*. 2015, pp. 52–57.

[287] C. Sadowski, E. Aftandilian, A. Eagle, L. Miller-Cushon, and C. Jaspan. “Lessons from building static analysis tools at Google”. In: *Commun. ACM* 61.4 (2018), pp. 58–66.

[288] M. Salnitri, F. Dalpiaz, and P. Giorgini. “Designing secure business processes with SecBPMN”. In: *Software & Systems Modeling* 16.3 (2017), pp. 737–757.

[289] Ó. Sánchez, F. Molina, J. García-Molina, and A. Toval. “ModelSec: A Generative Architecture for Model-Driven Security”. In: *JUCS - Journal of Universal Computer Science* 15.15 (2009), pp. 2957–2980. eprint: `https://doi.org/10.3217/jucs-015-15-2957`.

[290] P. Sánchez, A. Moreira, L. Fuentes, J. Araújo, and J. Magno. “Model-driven development for early aspects”. In: *Information and Software Technology* 52.3 (2010), pp. 249–273.

[291] R. Sandhu. “Lattice-based access control models”. In: *Computer* 26.11 (1993), pp. 9–19.

[292] F. Satoh, Y. Nakamura, and K. Ono. “Adding Authentication to Model Driven Security”. In: *2006 IEEE International Conference on Web Services (ICWS’06)*. 2006, pp. 585–594.

[293] S. Schefer-Wenzl and M. Strembeck. “Model-driven specification and enforcement of RBAC break-glass policies for process-aware information systems”. In: *Information and Software Technology* 56.10 (2014), pp. 1289–1308.

[294] F. B. Schneider. “Enforceable security policies”. In: *ACM Trans. Inf. Syst. Secur.* 3.1 (2000), pp. 30–50.

[295] S. Schneider, K. Kashish, K. Tuma, and R. Scandariato. "In Specs We Trust? Conformance-Analysis of Implementation to Specifications in Node-RED and Associated Security Risks". In: *Availability, Reliability and Security*. Springer Nature Switzerland, 2025, pp. 278–300.

[296] S. Schneider, K. Kashish, K. Tuma, and R. Scandariato. "In Specs We Trust? Conformance-Analysis of Implementation to Specifications in Node-RED and Associated Security Risks". In: *Availability, Reliability and Security*. Springer Nature Switzerland, 2025, pp. 278–300.

[297] A. Schürr and F. Klar. "15 Years of Triple Graph Grammars". In: *Graph Transformations*. Springer Berlin Heidelberg, 2008, pp. 411–425.

[298] M. Sefika, A. Sane, and R. Campbell. "Monitoring compliance of a software system with its high-level design models". In: *Proceedings of IEEE 18th International Conference on Software Engineering*. 1996, pp. 387–396.

[299] S. Seifermann. "Architectural Data Flow Analysis for Detecting Violations of Confidentiality Requirements". PhD thesis. Karlsruher Institut für Technologie (KIT) / Karlsruher Institut für Technologie (KIT), 2022. 282 pp.

[300] S. Seifermann, R. Heinrich, D. Werle, and R. Reussner. "Detecting Violations of Access Control and Information Flow Policies in Data Flow Diagrams". In: *Journal of Systems and Software* 184 (2022), p. 111138.

[301] *Semgrep*. Semgrep. URL: `https://github.com/semgrep/semgrep` (visited on 01/04/2026).

[302] S. M. A. Shah, K. Anastasakis, and B. Bordbar. "From UML to Alloy and back again". In: *Proceedings of the 6th International Workshop on Model-Driven Engineering, Verification and Validation*. MoDeVVa '09. Association for Computing Machinery, 2009.

[303] M. E. Shin and H. Gomaa. "Software requirements and architecture modeling for evolving non-secure applications into secure applications". In: *Science of Computer Programming* 66.1 (2007). Special Issue on the 5th International Workshop on System/Software Architectures (IWSSA'06), pp. 60–70.

[304] A. Shostack. *Threat Modeling: Designing for Security*. 1st. Wiley Publishing, 2014.

[305] S. Shuai, D. Guowei, G. Tao, Y. Tianchang, and S. Chenjie. "Modelling Analysis and Auto-detection of Cryptographic Misuse in Android Applications". In: *2014 IEEE 12th International Conference on Dependable, Autonomic and Secure Computing*. 2014, pp. 75–80.

[306] P. Shvaiko and J. Euzenat. "Ontology Matching: State of the Art and Future Challenges". In: *IEEE Transactions on Knowledge and Data Engineering* 25.1 (2013), pp. 158–176.

[307] M. L. Siddiq, N. Sekerak, A. Karam, M. Leal, A. Islam-Gomes, and J. C. S. Santos. *Assessing the Software Security Comprehension of Large Language Models*. 2025. arXiv: `2512.21238 [cs.SE]`.

[308] L. Singleton, R. Zhao, M. Song, and H. Siy. "CryptoTutor: Teaching Secure Coding Practices through Misuse Pattern Detection". In: *Proceedings of the 21st Annual Conference on Information Technology Education*. SIGITE '20. Association for Computing Machinery, 2020, pp. 403–408.

[309] G. Snelting, D. Giffhorn, J. Graf, C. Hammer, M. Hecker, M. Mohr, D. Wasserrab, and S. a. Bischof. *JOANA (Java Object-Sensitive Analysis) Project Page*. 2022. URL: https://pp.ipd.kit.edu/projects/joana/ (visited on 02/09/2026).

[310] Snyk Limited. *Snyk Website*. 2025. URL: https://snyk.io/ (visited on 02/09/2026).

[311] K. Solberg Söilen. *The Researcher's Journey: A Guide to Methodology and Academia in Social Sciences*. Springer Nature Switzerland, 2025.

[312] S. Soltan, M. Yannakakis, and G. Zussman. "REACT to Cyber Attacks on Power Grids". In: *IEEE Transactions on Network Science and Engineering* 6.3 (2019), pp. 459–473.

[313] SonarSource. *Security-Related Rules in SonarQube*.

[314] SonarSource. *SonarQube Check RSPEC-6384*. URL: https://rules.sonarsource.com/java (visited on 12/29/2025).

[315] *SpecificationTransfer*. URL: https://github.com/CASCADE-AnalysisCoupling/SpecificationTransfer (visited on 02/09/2026).

[316] H. Stachowiak. *Allgemeine Modelltheorie*. Springer, 1973.

[317] T. Stahl and M. Völter. *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons, Inc., 2006.

[318] W. Stallings. *Network and Internetwork Security: Principles and Practice*. Prentice-Hall, Inc., 1995.

[319] N. I. of Standards and Technology. *Minimum security requirements for federal information and information systems*. Tech. rep. National Institute of Standards and Technology (U.S.), 2006.

[320] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework*. Addison-Wesley Professional, 2008.

[321] C.-A. Sun, Y. Gong, M. Li, L. Xu, J. Han, and Y. Han. "Detecting Inconsistencies in Microservice-Based Systems: An Annotation-Assisted Scenario-Oriented Approach". In: *IEEE Transactions on Services Computing* 17.5 (2024), pp. 2194–2209.

[322] M. Swanson, J. Hash, and P. Bowen. *Guide for Developing Security Plans for Federal Information Systems*. NIST SP 800-18r1. National Institute of Standards and Technology, 2006, NIST SP 800–18r1.

[323] C. Talcott, S. Ananieva, K. Bae, B. Combemale, R. Heinrich, M. Hills, N. Khakpour, R. Reussner, B. Rumpe, P. Scandurra, and H. Vangheluwe. "Composition of Languages, Models, and Analyses". In: *Composing Model-Based Analysis Tools*. Springer International Publishing, 2021, pp. 45–70.

[324] C. Talcott, S. Ananieva, K. Bae, B. Combemale, R. Heinrich, M. Hills, N. Khakpour, R. Reussner, B. Rumpe, P. Scandurra, H. Vangheluwe, F. Durán, and S. Zschaler. "Foundations". In: *Composing Model-Based Analysis Tools*. Springer International Publishing, 2021, pp. 9–37.

[325] J. Tang, R. Li, H. Han, H. Zhang, and X. Gu. "Detecting Permission Over-claim of Android Applications with Static and Semantic Analysis Approach". In: *2017 IEEE Trustcom/BigDataSE/ICESS*. 2017, pp. 706–713.

[326] The Clang Team. *Clang Static Analyzer Security Checks*. 2025. URL: `https://clang.llvm.org/docs/ClangStaticAnalyzer.html` (visited on 02/09/2026).

[327] The MITRE Corporation. *Common Weakness Enumeration (CWE)*. URL: `https://cwe.mitre.org/` (visited on 02/09/2026).

[328] The MITRE Corporation. *CVE: Common Vulnerabilities and Exposures*. CVE. URL: `https://www.cve.org/` (visited on 01/12/2026).

[329] The MITRE Corporation. *CWE - CWE-20: Improper Input Validation (4.19)*. URL: `https://cwe.mitre.org/data/definitions/20#Vulnerability_Mapping_Notes_20` (visited on 02/09/2026).

[330] The MITRE Corporation. *CWE - CWE-489: Active Debug Code (4.18)*. URL: `https://cwe.mitre.org/data/definitions/489.html` (visited on 02/09/2026).

[331] The MITRE Corporation. *CWE - CWE-532: Insertion of Sensitive Information into Log File (4.19)*.

[332] The MITRE Corporation. *CWE - CWE-642: External Control of Critical State Data (4.19.1)*.

[333] The MITRE Corporation. *Vulnerability Metrics | CVE*. URL: `https://www.cve.org/about/Metrics` (visited on 02/09/2026).

[334] J.-P. Töberg, J. Schiffl, F. Reiche, B. Beckert, R. Heinrich, and R. Reussner. "Modeling and Enforcing Access Control Policies for Smart Contracts". In: *2022 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPS)*. 2022, pp. 38–47.

[335] A. Tolk. "Interoperability, Composability, and Their Implications for Distributed Simulation: Towards Mathematical Foundations of Simulation Interoperability". In: *2013 IEEE/ACM 17th International Symposium on Distributed Simulation and Real Time Applications*. 2013, pp. 3–9.

[336] L. Traub. "Kopplung von Architekturanalysen und musterbasierten Quelltextanalysen für Sicherheitseigenschaften von Aufrufabhängigkeiten". Final Thesis - Bachelor. Karlsruher Institut für Technologie (KIT) / Karlsruher Institut für Technologie (KIT), 2022. 62 pp.

[337] K. Tuma, S. Peldszus, D. Strüber, R. Scandariato, and J. Jürjens. "Checking security compliance between models and code". In: *Software and Systems Modeling* 22.1 (2023), pp. 273–296.

[338] K. Tuma, R. Scandariato, and M. Balliu. "Flaws in Flows: Unveiling Design Flaws via Information Flow Analysis". In: *2019 IEEE International Conference on Software Architecture (ICSA)*. 2019, pp. 191–200.

[339] B. Uzun and B. Tekinerdogan. "Architecture Conformance Analysis Using Model-Based Testing: A Case Study Approach". In: *Software: Practice and Experience* 49.3 (2019), pp. 423–448.

[340] A. V. Uzunov, E. B. Fernandez, and K. Falkner. "Engineering Security into Distributed Systems: A Survey of Methodologies". In: *JUCS - Journal of Universal Computer Science* 18.20 (2012), pp. 2920–3006.

[341] N. Vachharajani, M. Bridges, J. Chang, R. Rangan, G. Ottoni, J. Blome, G. Reis, M. Vachharajani, and D. August. "RIFLE: An Architectural Framework for User-Centric Information-Flow Security". In: *37th International Symposium on Microarchitecture (MICRO-37'04)*. 2004, pp. 243–254.

[342] M. H. Valipour, B. Amirzafari, K. N. Maleki, and N. Daneshpour. "A brief survey of software architecture concepts and service oriented architecture". In: *2009 2nd IEEE International Conference on Computer Science and Information Technology*. 2009, pp. 34–38.

[343] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. "Soot: a Java bytecode optimization framework". In: *CASCON First Decade High Impact Papers*. CASCON '10. IBM Corp., 2010, pp. 214–224.

[344] C. Vassallo, F. Palomba, A. Bacchelli, and H. C. Gall. "Continuous code quality: are we (really) doing that?" In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ASE '18. Association for Computing Machinery, 2018, pp. 790–795.

[345] R. D. Venkatasubramanyam and S. G. R. "Why is dynamic analysis not used as extensively as static analysis: an industrial study". In: *Proceedings of the 1st International Workshop on Software Engineering Research and Industrial Practices*. SER&IPs 2014. Association for Computing Machinery, 2014, pp. 24–33.

[346] H. Wada, J. Suzuki, and K. Oba. "A Model-Driven Development Framework for Non-Functional Aspects in Service Oriented Grids". In: *International Conference on Autonomic and Autonomous Systems (ICAS'06)*. 2006, pp. 30–30.

[347] N. Walkinshaw, M. Roper, and M. Wood. "The Java system dependence graph". In: *Proceedings Third IEEE International Workshop on Source Code Analysis and Manipulation*. 2003, pp. 55–64.

[348] M. Walter, R. Heinrich, and R. Reussner. "Architectural Attack Propagation Analysis for Identifying Confidentiality Issues". In: *2022 IEEE 19th International Conference on Software Architecture (ICSA)*. 2022, pp. 1–12.

[349] H. Wang, Y. Guo, Z. Tang, G. Bai, and X. Chen. "Reevaluating Android Permission Gaps with Static and Dynamic Analysis". In: *2015 IEEE Global Communications Conference (GLOBECOM)*. 2015, pp. 1–6.

[350] J. Wang and Y. Chen. "A Review on Code Generation with LLMs: Application and Evaluation". In: *2023 IEEE International Conference on Medical Artificial Intelligence (MedAI)*. 2023, pp. 284–289.

[351] Z. Wang, L. Sun, and H. Zhu. "Defining Social Engineering in Cybersecurity". In: *IEEE Access* 8 (2020), pp. 85094–85115.

[352] A. Wasowski and T. Berger. *Domain-Specific Languages – Effective Modeling, Automation, and Reuse*. Springer International Publishing, 2023.

[353] F. Wei, S. Roy, X. Ou, and Robby. "Amandroid: A Precise and General Inter-component Data Flow Analysis Framework for Security Vetting of Android Apps". In: *ACM Trans. Priv. Secur.* 21.3 (2018).

[354] D. A. Wheler. *Flawfinder*. 2001. URL: `https://dwheeler.com/flawfinder/` (visited on 02/10/2026).

[355] C. Wolter, M. Menzel, A. Schaad, P. Miseldine, and C. Meinel. "Model-driven business process security requirement specification". In: *Journal of Systems Architecture* 55.4 (2009). Secure Service-Oriented Architectures (Special Issue on Secure SOA), pp. 211–223.

[356] N. Wongpakaran, T. Wongpakaran, D. Wedding, and K. L. Gwet. "A Comparison of Cohen's Kappa and Gwet's AC1 When Calculating Inter-Rater Reliability Coefficients: A Study Conducted with Personality Disorder Samples". In: *BMC Medical Research Methodology* 13.1 (2013).

[357] L. Xiao. "An adaptive security model using agent-oriented MDA". In: *Information and Software Technology* 51.5 (2009). SPECIAL ISSUE: Model-Driven Development for Secure Information Systems, pp. 933–955.

[358] D. Xu and K. Nygard. "Threat-driven modeling and verification of secure software using aspect-oriented Petri nets". In: *IEEE Transactions on Software Engineering* 32.4 (2006), pp. 265–278.

[359] D. Xu, M. Tu, M. Sanford, L. Thomas, D. Woodraska, and W. Xu. "Automated Security Test Generation with Formal Threat Models". In: *IEEE Transactions on Dependable and Secure Computing* 9.4 (2012), pp. 526–540.

[360] D. Xu, W. Xu, M. Kent, L. Thomas, and L. Wang. "An Automated Test Generation Technique for Software Quality Assurance". In: *IEEE Transactions on Reliability* 64.1 (2015), pp. 247–268.

[361] Z. Xu, X. Hu, Y. Tao, and S. Qin. "Analyzing Cryptographic API Usages for Android Applications Using HMM and N-Gram". In: *2020 International Symposium on Theoretical Aspects of Software Engineering (TASE)*. 2020, pp. 153–160.

[362] H. Yu, D. Liu, X. He, L. Yang, and S. Gao. "Secure software architectures design by aspect orientation". In: *10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'05)*. 2005, pp. 47–55.

[363] K. Yurchenko, M. Behr, H. Klare, M. Kramer, and R. Reussner. "Architecture-driven reduction of specification overhead for verifying confidentiality in component-based software systems". In: *ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS 2017), Austin, TX, September 17-22, 2017. Ed.: L. Burgueño.* Vol. 2019. CEUR Workshop Proceedings. 2017, pp. 321–323.

[364] J. Zhang, J. Zheng, Z. Zhang, T. Chen, Y.-a. Tan, Q. Zhang, and Y. Li. "ATT&CK-based Advanced Persistent Threat attacks risk propagation assessment model for zero trust networks". In: *Computer Networks* 245 (2024), p. 110376.

[365] T. Zhang, W. Shen, D. Lee, C. Jung, A. M. Azab, and R. Wang. "PeX: a permission check analysis framework for Linux kernel". In: *Proceedings of the 28th USENIX Conference on Security Symposium*. SEC'19. USENIX Association, 2019, pp. 1205–1220.

[366] Y. Zhang, M. M. A. Kabir, Y. Xiao, D. Yao, and N. Meng. "Automatic Detection of Java Cryptographic API Misuses: Are We There Yet?" In: *IEEE Transactions on Software Engineering* 49.1 (2023), pp. 288–303.

[367] Z. Zhioua, S. Short, and Y. Roudier. "Static Code Analysis for Software Security Verification: Problems and Approaches". In: *2014 IEEE 38th International Computer Software and Applications Conference Workshops*. 2014, pp. 102–109.

[368] C. Zhong, H. Zhang, H. Huang, Z. Chen, C. Li, X. Liu, and S. Li. "DOMICO: Checking conformance between domain models and implementations". In: *Software: Practice and Experience* 54.4 (2024), pp. 595–616.

[369] Z. J. Zhu and M. Zulkernine. "A model-based aspect-oriented framework for building intrusion-aware software systems". In: *Information and Software Technology* 51.5 (2009). SPECIAL ISSUE: Model-Driven Development for Secure Information Systems, pp. 865–875.