Detecting Information Flow Security Vulnerabilities by Analysis Coupling

Frederik Reiche, Ralf Reussner, Robert Heinrich

Abstract—Security vulnerabilities originating from insecure information flows can violate the confidentiality of data, thereby negatively impacting individuals and service providers. This challenge gave rise to design-level analyses and source code analyses investigating information flow-related vulnerabilities. Architectural analysis, a type of design-level analysis, can detect security vulnerabilities by inspecting architectural models enriched with specifications of security-relevant information. However, the implementation may not comply with the architectural specification during software evolution. This non-compliance can result in the architectural analysis missing vulnerabilities. Consequently, vulnerabilities in the deployed system can be exploited, but the software engineers are left assuming the system to be secure. In this article, we address this problem of specification-related non-compliance by proposing a coupling approach that enables architectural analyses to use the values of security characteristics which are supplied from the implementation and retrieved by static source code analysis. Our coupling approach makes two contributions: a coupling process and the conditions necessary for the coupling (called integration conditions). In our coupling process, each process step performs transformations between the involved input and output models of the analyses. To enable the coupling, we define necessary integration conditions that must hold between the (meta)models of the analyses in the coupling. We generalize from specific analyses by specifying the integration conditions based on reference metamodels. In our evaluation, we inspect (1) the coverage of the reference metamodels by the metamodels of coupled analyses, (2) the coverage of the integration conditions by successful couplings, and (3) the accuracy of the coupled analysis in finding architectural vulnerabilities originating from a non-compliant implementation. The results of our case study show that the reference metamodels and the integration conditions are covered. We detect 60 true positive vulnerabilities and 5 false positive vulnerabilities. Upon this evidence, we conclude that the architectural analysis in the coupling is accurate in detecting vulnerabilities originating from non-compliant information flows in the implementation.

Index Terms—Security Analysis, Architecture Implementation Consistency, Composition, Model-driven Engineering, Security-by-Design.

1 Introduction

The number of security vulnerabilities reported in software systems has risen in recent years [1]. A *security vulnerability* is a known weakness in the system (e.g., in its design or implementation) that an actor can exploit [2]. Exploitation of vulnerabilities can compromise the confidentiality of data processed in software systems. Confidentiality is a critical security objective, and its violation negatively impacts individuals and service providers, as exemplified by the Equifax data breach [3]. When a confidentiality vulnerability is exploited, individuals need to worry that their data is being misused (e.g., for impersonation [4]) and service providers face losses in financial resources and in reputation [3], [4], [5]. Even worse, confidentiality vulnerabilities in safety-critical systems (e.g., mobility or energy supply) present particularly alarming cases for misuse.

Confidentiality vulnerabilities tend to follow the data flow [6], because confidentiality is often violated by exploiting vulnerabilities arising in data flows. Research also shows that confidentiality can be compromised by the control flows [7], [8]. Therefore, vulnerabilities in information flow (i.e., the combination of data flow and control flows [7]) deserve special and *early* attention for purposes of detection and fixing. Many works on detection and fixing of vulnerabilities have investigated the static analysis of information flows in the implementation phase (e.g., [8], [9], [10], [11], [12]). However, the later an issue in the system is fixed, the more cost-intensive the fix becomes [13]. Consequently, every detection and fix should happen during the development phase, when issues — or in our case, *vulnerabilities* — are introduced [14]. Tackling issues early is especially pertinent to vulnerabilities introduced in the design phase, because design flaws cause approximately 50% of all vulnerabilities [15] but also because *Insecure Design* ranks Number 4 in the Top Ten Security Risks of the Open Web Application Security Project (OWASP) [16]. For these reasons, it is essential that design vulnerabilities are detected and fixed before the implementation. In fact, system design already is a focus of security analysis research, yielding approaches like UMLSec [17].

Many design-level security analyses have an input model that comprises the system design and security specification, e.g., [6], [17], [18], [19], [20], [21], [22]. The system design represents abstractions of the structure and behavior of the system. The security specification enriches the system design with security characteristics [14] relevant to the detection of vulnerabilities. Likewise, many source code information flow analyses, e.g., [10], [12], [23], [24], [25], use the implementation and security specifications which describe information-flow-related security characteristics that are expected to hold due to the information flows in the system [7], [26]. Security characteristics are represented by a set of values and describe security-relevant properties of the system or its elements (e.g., confidentiality levels of data).

Ideally, once design-level analyses have determined a

• *Frederik Reiche and Ralf Reussner are with Karlsruhe Institute of Technology, Karlsruhe, Germany. Robert Heinrich is with Ulm University, Ulm, Germany.*
E-mail: {frederik.reiche,reussner}@kit.edu, robert.heinrich@uni-ulm.de

system to be free of vulnerabilities, it will be implemented according to the system design and security specification. However, the implementation can fail to comply with the design or specifications [27], [28], [29] because of repeated evolutionary changes or implementation errors like missing encryption [28] or unexpected information flows [29]. We call the failure to comply with system design *design-related non-compliance*, while we call the failure to comply with security specification *specification-related non-compliance*. Design-related non-compliance occurs when the implementation does not realize the given system design, e.g., by not encrypting data despite being designed to [28].

Specification-related non-compliance occurs when the implementation does not realize a given specification. As example, imagine a service parameter in the design-level analysis input model being specified to contain only non-confidential data by a security characteristic with the value *non-confidential*. When a source code analysis detects an information flow of *confidential* data to the corresponding parameter in the implementation, the parameter will contain confidential data and consequently fail to comply with the specified value. The information detected here — revealed by the source code analysis — is in fact the values of a security characteristic that manifest in the implementation. (We refer to such information as the *values resulting from the implementation*, which is further defined in Section 2.)

The above example makes clear that specification-related non-compliance may arise when the implementation fails to comply with system design. In the example, when the encryption is not implemented as designed, the confidentiality level of the data is also not lowered as expected in the specification. Conversely, though, the design-related non-compliance does not necessarily arise if a specification-related non-compliance exists, e.g., when the analysis uses the specification to abstract information like behavior description. This is a crucial point, because it means that specification-related non-compliance can be more difficult for software engineers to detect since the cause may be wholly unrelated to the system design. In this connection, it is worth noting that non-compliances can be caused, amongst others, by software engineers who are not always security experts [30]. For this reason, the security must still be ensured in later stages of development, such as the implementation phase, even when performing design-time security analysis [6].

The upshot of it all is this: When a design-level analysis is performed without information about the implementation, then vulnerabilities will be identified *only upon the assumption that* the implementation complies with the design and specification [28], [31]. As a consequence, specification-related non-compliance can lead to Problem **P: False Design-level Analysis Results because of Specified Values of Security Characteristics not Reflecting Values Realized by the Implementation** (illustrated in Section 3). In Problem P, the design-level analysis cannot detect a vulnerability of the implemented system because the analysis uses values of security characteristics specified by software engineers in the design phase, rather than using values of security characteristics resulting from the implementation. As consequence of Problem P, software engineers expect the system to be secure because the design-level analysis does not report vulnerabilities while attackers can exploit confidentiality vulnerabilities when the system is deployed and operating.

In this article, we address Problem P by coupling an architectural analysis and a static source code information flow analysis: our aim in this is to enrich the architectural analysis by the values of security characteristics that result from the implementation. Notably, our coupled analyses are executable without deploying a potentially insecure system. Our coupling is described on architectural security analysis, which is a subclass of security analysis in system design because architectural design decisions influence quality properties such as security [32]. More precisely, we focus on architectural security analyses with security characteristics in their input model that are influenced by the information flow. The input model of the architectural analysis comprises design decisions made due to information from prior development stages. Thus, we assume the input model to be prescriptive [33], and that it also correctly captures information from prior stages, such as threat modeling or stakeholder requirements. In this way, we exclude cases either where the specification does not enable building a secure system or the implementation is defined as correct.

Our coupling is also described for static source code information flow analyses that use specifications of security characteristics of data, such as [10], [20], [23], [24], [25]. Source code analyses closely resemble model-based analyses because the implementation of a system can be interpreted as a model [34]. Furthermore, security characteristics of data are often specified for architectural security analyses like [6], [17], [18], [19], [20], [21], [22], [29], [35], as these characteristics are directly influenced by the information flow.

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. Note that our approach does not counter the intuitive understanding that architectural design activities are performed prior to the implementation. Still, architectural analyses can be performed without the availability of implementation artifacts. However, to avoid the need for using values of security characteristics specified by software engineers in the design phase, our analyses can be performed once the implementation is available to strengthen the results of the architectural analyses. Therefore, our coupling approach considers an incremental development scenario in which an architectural design and implementation are already available. It is important to understand that architectural analyses and design models in incremental development scenarios are still relevant even when an implementation is available because 1) they detect vulnerabilities based on information not available in the implementation, such as attackers [19], [21], [36], timing information [37] or deployment [14], [18], [21], [22] and 2) they have to be executed in every evolution step when the architecture is changed to avoid implementing a system in which architectural vulnerabilities exist [28]. Design models are also usable when the implementation is available to guide other security-related tasks, like executable tests [38], [39] and penetration testing [40], [41]. We do not discuss these approaches further as they are not in the scope of this article.

The coupling enables the following two benefits for software engineers: (**B1**) Software engineers are aware of architectural vulnerabilities arising from a non-compliant

implementation that would be missed when performing the architectural analysis in isolation, as illustrated in the running example in Section 3. (**B2**) Section 3 also demonstrates a situation where the architectural analysis does not detect new vulnerabilities despite enriching it with values resulting from the non-compliant implementation. Consequently, the coupling enables software engineers to distinguish the effect of a non-compliant implementation on the architectural analysis results. Insight into these effects is important as achieving complete security is usually not feasible due to resource limitations [42], [43], requiring prioritization of security vulnerabilities to be fixed.

The coupling approach presented in this article to address Problem P comprises the following contributions:

- **C1** We provide a systematic process for coupling architectural analyses and source code analyses. This process enables us to enrich the architectural analysis input model with the values of security characteristics resulting from the implementation to detect vulnerabilities by the architectural analysis that cannot be detected when performing the architectural analysis in isolation. For each process step, we describe the input and output models involved in the analyses, transformations between these models, and the responsibilities of roles. In contrast, related work such as [18], [29], [44], [45], [46] does not consider the impact of a non-compliant implementation on the result of the architectural analyses. With the coupling process, we achieve benefit **B1** and **B2**.
- **C2** We describe necessary relations between the input and output models of the architectural analysis and source code analysis to establish a coupling. These relations (named *integration conditions* in this article) ensure that the integration of the values of a security characteristic resulting from the implementation into the architectural analysis is possible. The relations are described independently of specific models, analyses, and security characteristics. To achieve this independence, we formulate the relations based on abstractions of the input and output (meta)models of the analyses captured in reference metamodels. In contrast, related work such as [18], [28], [29], [44] provides approaches tailored to specific models, analyses, and security characteristics. Therefore, engineers must extract knowledge about the coupling from related work and transfer it without further guidance to create couplings for other architectural analyses and source code analyses. With the proposed reference metamodels and integration conditions, we provide reusable knowledge for engineers that can be applied in coupling various architectural and source code analyses using our coupling approach.

This article is structured as follows: Section 2 provides terms and definitions. Section 3 provides a running example. An overview of the coupling approach is provided in Section 4. We describe the coupling process in Section 5. Section 6 presents the reference metamodels. We describe the integration conditions in Section 7. Section 8 discusses efforts arising in the coupling process. Section 9 presents the evaluation design and the results. Related work is discussed in Section 10. Limitations and future work are discussed in Section 11. Section 12 concludes the article.

2 Terms and Definitions

This section describes the terms and definitions necessary for this article. The coupling approach couples *model-based analyses* using model-driven software engineering techniques. Consequently, we introduce terms from model-driven software engineering in Section 2.1 and *Model-based Security Analyses* in Section 2.2. We introduce the notion of non-compliance between architectural models and implementation addressed in our coupling approach in Section 2.3. In Section 2.4, we differentiate available coupling processes and define the term of *coupling scenario* necessary to understand the reasoning for our selected coupling approach. Finally, a coupling is only useful when the exchanged information is useful for the analyses on the *syntactic* and *semantic* level [47], [48]. We introduce these terms in Section 2.5 as we have to address them in a comprehensive coupling approach.

Horizontal bars indicate definitions we provide in this article specifically for our approach that are not found in the literature, such as refinements of existing definitions.

2.1 Model-Driven Software Engineering

In model-driven software engineering, models are the main artifacts to be processed. A model is an abstraction of an entity for a given purpose with a subset of its attributes [49]. The structure of a *model* is defined by a *metamodel*, comprising allowed *metaclasses* and directed relations between the *metaclasses* called *reference* [50]. Model *transformations* translate one or more input models into one or more output models [51]. The *transformation* between elements of the input and output models is described by *transformation rules* [50]. The *transformation rules* are described based on *correspondences* between the elements of the input and output metamodels, defining a relation between them [52]. *Correspondences* can be captured in a *correspondence model*, which is based on a *correspondence metamodel* [53].

2.2 Model-Based Security Analysis

A *model-based analysis* evaluates *input models* with a certain *analysis technique* to provide an answer to a given *question* in the form of *output models* [54]. As our coupling approach focuses on the coupling of architectural analyses that use information flow-related security characteristics of data in architectural models with static source code analyses investigating information flows in the implementation, we call these analyses only *architectural analyses* and *source code analyses*. We use the term *information flow analysis* also for data flow analysis as both investigate the data flow, while the former also investigates the control flow. The syntax of the input models and output models of *model-based analyses*, i.e., the available model elements and relations between them, can be described by metamodels or grammars. We focus on analyses that use models [50] because we build on our preliminary work on metamodel evolution for model-based analyses [55]. For illustrative purposes, we simplify by assuming that the input and output models of the analyses are each represented in a single model. However, we are aware that the input and output of an analysis can be represented by several models and files [55].

Input Models: The input models of specification-based analyses comprise system representations and specifications. Component-based architectural models or the implementation are examples of system representations. The specifications define expected *Security Characteristics* for system elements, e.g., on the architecture or implementation level. A security characteristic comprises a type and a set of values to describe security-relevant properties in a representation of the system or its elements, such as an architectural model [14]. We simplify in this article by interpreting the values as instances of a security characteristic metaclass. This is realized in different model-based analyses such as [18], [21], [22], [23], [29].

Security characteristics describe a wide variety of security-relevant properties: Hahner et al. [35] use security characteristic with the values *EU* and *Non-EU* to describe the location of resources (European Union or not). The analysis of Kramer et al. [21] uses security characteristics to describe protection mechanisms against tampering with resources by values such as *Seal* or *Special Screws*. UMLSec [17] describes the protection of communication links, e.g, with values *encrypted* or *LAN*. *Security characteristics of data* are an often used type of security characteristic in architectural security analyses and source code information flow analyses. Examples often used in information flow analysis are roles or confidentiality levels. The CANUKUS example by Bell [56] specifies the role an actor must have to be allowed to obtain the data, e.g., by the basic values *UK* (United Kingdom), *US* (United States) or the combination of them, e.g., *UKUS*. Basic values can be interpreted 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 in information flow literature [57].

The available values of security characteristics in an analysis input model are either *predefined* by the analysis (e.g., [19], [37]) or *specifiable* by the analysis user (e.g., [6], [18], [20], [21], [23], [24]). Specifications are attached to system representations by various mechanisms [55], e.g., stereotypes in UMLSec [17], Java annotations in *Java Object-sensitive ANAlysis (JOANA)* [23] or textual representations in FlowDroid [12]. We subsume all these mechanisms under the term *annotation*. Annotations are a common mechanism to provide information to models representing the system [55].

Question: The *question* in security analysis commonly asks whether a vulnerability exists in the system. This question is answered by examining whether an analysis input model violates a given *security policy* [58]. 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., [8], [19], [23], [29], or architectural analysis, such as [14], [19], [21], [35]. Therefore, we provide Definition 1 for security policies considering security characteristics.

Definition 1 (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.

A common *security policy* in information flow analysis is a *lattice*, which describes allowed or unallowed information flows between values of a security characteristic of data [8].

When using security characteristics of data representing roles, the *lattice* allows to describe whether the combination of roles is interpreted *conjunctive* or *disjunctive*. In the *disjunctive* case, as in the CANUKUS example of Bell [56], 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 *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.

Analyses like [6], [18], [23], [29] provide security policies that are *configurable* by the user of the analyses, i.e., they allow the specification of the relations between the values of security characteristics in security policies. In contrast, analyses like [12], [19], [21], [37] use an *integrated* security policy that uses predefined rules to define the relation between the values of security characteristics.

Output Models: The output models represent the answer to the *question*. They are an abstraction of the input models [48], e.g., in the form of counter-examples [54].

2.3 Non-Compliance between Implementation and Architectural Specification

Tuma et al. [28] define an implementation to comply with the architectural model if a source code analysis does not report a vulnerability in the system for behavioral information derived from the design. We adapt this definition by defining non-compliance of the implementation with an architectural specification regarding values of security characteristics specified in the architectural specification in Definition 2.

Definition 2 (Non-Compliant Values of Security Characteristics between Implementation and Architectural Specifications). Given an architectural model containing security characteristics for architectural elements and a source code analysis that analyzes the implementation and 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 an architectural element if the values of the security characteristics derived for an implementation element cannot be mapped to the values specified for a corresponding architectural element.

2.4 Composition of Analyses

In model-based analysis composition, the different models and analysis tools are the primary artifacts that are composed or decomposed to achieve a composition goal [48]. There are three types of composition: *white-box*, *grey-box* and *black-box* [48]. In *white-box* composition, the metamodels of analyses are combined. In this composition type, the internals of the metamodels and analysis tools are open for modification. In *grey-box* composition, the steps of two or more *analysis algorithms* are coordinated to compute a result.

Typically, a coupling framework, such as OPAL [59], provides functionality that has to be introduced into the analysis for synchronization and communication.

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

An *Analysis Coupling* approach provides the means to connect two analyses to achieve a specific type of *Analysis Composition*. A coupling is realized based on a given coupling scenario, specified in Definition 3. The coupling scenario in this article focuses on specification-related non-compliance according to Definition 2.

Definition 3 (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.

In this article, we denote an analysis comprised in a coupling scenario as *analysis in the coupling* and an analysis applied without coupling as an *isolated analysis*. The realization of a coupling scenario yields *coupled analyses*.

2.5 Syntax and Semantics

The syntax of models is defined by their metamodels. However, the syntax itself does not convey the meaning of *syntactic elements*. A metaclass *lock* can have the meaning of a security mechanism (door lock) or a synchronization mechanism in parallel computing. This is why the *syntactic elements* of (meta)models must be mapped to *semantic elements* of a *semantic domain*, which describes concepts of a given universe of discourse [60]. This mapping is called a *semantic mapping* [54], [60].

The information exchange in a coupling is realized on the *syntax level*. However, the coupling is likely to be invalid if the coupled analyses interpret exchanged information differently, i.e., they use different *semantics* [48]. Determining the semantics in a coupling of model-based analysis can be challenging when only considering the metamodel level, as illustrated with the following example: The input metamodel of a source code analysis comprises the metaclass *Security Level*, which can only be instantiated to the values *high* and *low*. The input metamodel of the architectural analysis comprises the metaclass *Security Level*, where the values can be defined freely. Both metaclasses can be determined to correspond on the metamodel level as they describe a *Security Level*. Still, on the model level, this correspondence is only valid if the *Security Level* of the architectural analysis is instantiated to the values *high* and *low*. We call this capability of metaclasses *model-level semantic refinement*. While we address metamodels and models, this capability is analogous to the semantic refinement between a reference model and a specific model defined by Konersmann et al. [61].

3 Running Example

Our running example consists of a sample system comprising an architectural design and implementation, an example architecture-based confidentiality analysis investigating security characteristics influenced by information flows, and an example static source code analysis for information flow. All models in our running example are illustrated in Fig. 1. Partial models (a) to (f) mark the input and output models of the analyses, and (1) to (15) mark locations of interest.

3.1 System

We adopt an online shop system [35] to illustrate our coupling approach.

Architecture: The architecture of the online shop system is described using the Palladio Component Model (PCM) [62], an Architecture Description Language (ADL) for component-based architectures. In addition to the system description, Hahner et al. [35] define the confidentiality levels for data (security characteristic of data) that can be influenced by implemented information flows, making the system usable as a running example. The architectural design of the system is illustrated in partial model (a) of Fig. 1. The system consists of the components *OnlineShop* and *DataStorage*. The *OnlineShop* provides the *buy* service to buy items and uses the *store* service of the *DataStorage* to store *UserData*. The *buy* service contains the parameters *userData* for the representation of the user data and *cart* for the representation of the items a user wants to buy. The *store* service receives *userData* to be stored. The components *OnlineShop* and *DataStorage* are allocated to the resources *On-Site* and *Cloud*, respectively.

Implementation: The implementation of the *OnlineShop* and *DataStorage* components is illustrated in partial model (b) of Fig. 1. The implementation of the *OnlineShop* includes the field *privilegeSum* and the implementation of the *buy* service in the architectural model in a correspondingly named method. The implementation of the *DataStorage* includes an implementation of the *store* service in the architectural model in a correspondingly named method. We distinguish the architectural and source code levels using the terms service and method, respectively. The field *privilegeSum* determines whether a user gets a privileged status. An *if* statement in the *buy* method compares the value of the items in the cart (*cart.sum()*) to the value of *privilegeSum* and realizes the following behavior: If the value of the items in the cart is higher than the *privilegeSum*, the *userData* is modified by setting the *privilege* field to *true*. The modification is then stored by calling the *store* method of *DataStorage*.

3.2 Architectural Confidentiality Analysis

We analyze architectural vulnerabilities by applying an architectural analysis similar to the Data Flow Analysis (DFA) of Seifermann et al. [6]. The architectural analysis checks the secure deployment of components on resources based on

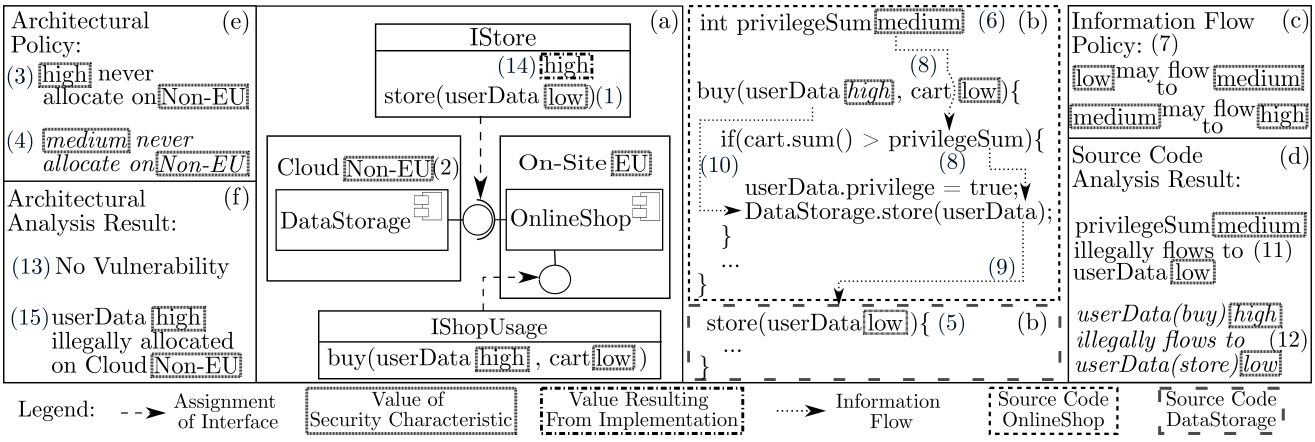

Fig. 1. Combined diagrams showing (a) parts of the architectural model and (b) parts of the implementation of the components OnlineShop and DataStorage for the Online Shop System with security specifications. (c) Shows the information analysis policy and (d) the calculated illegal flows. (e) Shows the applied architectural policies and (f) the architectural analysis results before and after inserting the values resulting from the implementation. Italic font marks those elements that are altered in an alternative scenario to illustrate benefit B2. Adapted from Hahner et al. [35].

the architectural design model and security specifications illustrated in partial model (a) of Fig. 1. We adapt the example of Hahner et al. [35] to illustrate our approach. The confidentiality level of data (security characteristic of data) is represented by the values *high*, *medium* (not represented in partial model (a)), and *low*. These values are specified for *parameters* of services to define whether they are expected to contain highly confidential (*high*), confidential (*medium*), or public (*low*) data. We specify locations of *resource container* in the architectural model by the security characteristic *location* with the values *EU* and *Non-EU*. In Fig. 1, the specification of locations and confidentiality levels are presented by their values for resources and parameters respectively. For example, the parameter *userData* of the *store* service is specified to contain *low* data (1) and the resource *Cloud* is defined to be located in *Non-EU* (2). Based on the values of the security characteristics, we use a security policy illustrated in partial model (e) of Fig. 1. This policy defines that data, marked with the confidentiality level *high* (3) or *medium* (4), must not be used in components allocated to resources with the *location Non-EU*. This analysis is usable in our running example as the confidentiality level, specified for parameters of the services, is influenced by the implemented information flows illustrated in partial model (b) of Fig. 1.

3.3 Static Source Code Analysis for Information Flow

We analyze the information flows in this example using *JOANA* [23] as source code analysis. *JOANA* is already used in related work [29] to analyze the compliance between design models and the implementation. For illustration, we describe a simplified version of *JOANA*, where the security characteristic of data represents confidentiality levels with the values *low*, *medium*, and *high* specified for *fields* and *parameters* in the source code. For example, the parameter *userData* in the *store* method is assigned the value *low* (5), and the field *privilegeSum* in the class *OnlineShop* is assigned the value *medium* (6). JOANA uses the information flow policy (7), illustrated in partial model (c) of Fig. 1, to analyze whether information flows from implementation elements with a higher confidentiality level to others with lower ones exist. These flows indicate a vulnerability.

3.4 Illustration of the Problem

The specification in the architectural model expects the *userData* in the *store* service to contain information with the confidentiality level *low*. When performing the architectural analysis for the designed system with the security policy in partial model (e) of Fig. 1, it reports no vulnerability (13) regarding the specified values *low* of the *userData* (1) in the architectural model. However, in the implementation, JOANA detects the two unallowed information flows (11) and (12) presented in the source code analysis output (illustrated in partial model (d) of Fig. 1). Information of the confidentiality level *medium* is leaked to *userData* of the *store* method, as the call to the *store* method (9) depends on whether the total value of *cart* is larger than the *medium* classified *privilegeSum* (8). Information of the confidentiality level *high* is leaked to *userData* of the *store* method because the *high* classified *userData* of the *buy* method (10) is passed to the *userData* of the *store* method (9) classified as *low* (5). Through these flows, *userData* in the *store* method contains data with the confidentiality level *medium* or *high* (14), which does not comply with the specified value *low* in the architectural model (1). When considering these values for the *userData* in the *store* service, the architectural analysis detects a vulnerability (15) as confidential data is processed in the *Cloud* resource specified with location *Non-EU*.

3.5 Illustration of the Benefits of the Coupling

The coupling approach proposed in this article aims to enable the usage of the information about the non-compliant value *high* resulting from the implementation (14) in the architectural analysis. With this approach, the architectural analysis detects the vulnerability that *high* confidential data is processed on the *Cloud* resource located in *Non-EU*. This situation illustrates benefit **B1**, where the software engineer is aware of architectural security vulnerabilities that arise from the implementation and would not be identified by the architectural analysis based on the values specified in the design phase. For illustrating benefit **B2**, we exemplarily modify the running example as follows: We remove requirement (4) from the security policy of the architectural analysis. This

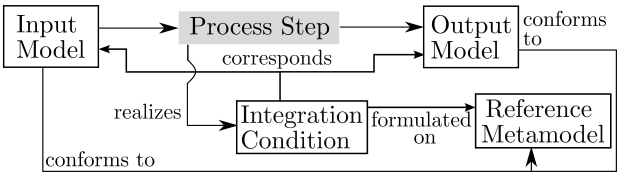

Fig. 2. Overview of the relation between models of the analyses, integration conditions and reference metamodels. See Fig. 3 for legend.

requirement states that data with the confidentiality level *medium* must not be processed by components allocated to resources located in *Non-EU*. Consequently, the architectural analysis only reports vulnerabilities if data with confidentiality level *high* are processed on such resources. In addition, the *userData* in the service *buy* contains only *low* information. Consequently, illegal flow from *privilegeSum* to *userData* in the *store* method ((9) and (10)) is removed, also removing its report (11) in the source code analysis result. As a result, the value resulting from the implementation for *userData* in the *store* service is now *medium*. In this modified scenario, the value resulting from the implementation of *userData* is non-compliant with the architectural specification (i.e., *medium* instead of *low*), but the architectural analysis reports no vulnerability (13). This illustrates benefit **B2** as software engineers can identify whether non-compliance, reported by a source code analysis, affects the architectural security.

4 Overview of the Coupling Approach

Our coupling approach comprises a *Coupling Process*, *Integration Conditions* and *Reference Metamodels*. Their relation are shown in Fig. 2. The coupling process describes process steps that must be performed for each coupling independent of the specific analyses. At the same time, the metamodels and models of the analyses have to be validated against the Integration Conditions (ICs) for each individual coupling. This is why we separate the description of the coupling process in Section 5 from the description of the ICs in Section 7 to highlight the modularity of our approach and to address the separated contributions **C1** and **C2**. We also provide a discussion about the novelty of our approach in Section 4.1 which is later complemented with discussions of related work in Section 10.

Coupling Process: The *coupling process* (which is part of **C1**) describes how the coupling is achieved. The coupling process enriches the architectural analysis input model with values of security characteristic resulting from the implementation by applying a source code analysis as defined in a *coupling scenario* (see Definition 3). Therefore, with this coupling process, the architectural analysis is no longer performed upon the assumption that the implementation complies with the values of the security characteristic specified in the architectural analysis input model.

For the coupling process, we apply a black-box coupling (see Section 2.4). We motivate the choice for a black-box coupling and discuss its benefits in the coupling of architectural analyses and static source code analyses in Section 4.2. In the remainder of this article, we focus on the input and the output of the architectural analysis and source code analysis in the form of metamodels and models as they are the artifacts used in a black-box coupling.

Creating a coupling between analyses is a challenging task [48] because knowledge about the single analyses and the connections between them is required. We address the complexity in this task by assigning responsibilities and necessary competencies to developer roles which allows separating concerns in applying the analysis coupling process in a development team. We describe these roles in Section 4.3.

Integration Conditions: The black-box coupling introduces the challenge that the information captured in the models must suffice to integrate the values of the security characteristic resulting from the implementation into the architectural model. However, descriptions of generalized transformations in the coupling process steps for all existing architectural and source code analyses may not be possible due to differences in the syntax and semantics of their input and output models. For instance, the source code analysis Jflow [10] requires the specification of the values of the security characteristic of data for implementation elements, such as parameters or fields. In contrast, the source code analysis FlowDroid [12] requires the specification of sources and sinks. Bug-finding analysis [26] is a different type of security analysis that does not use any specifications and provides locations of found bug patterns in its output. Our contribution **C2** addresses this challenge by providing ICs, detailed in Section 7, that must hold so that integrating the values of the security characteristic of data resulting from the implementation into the architectural specification is possible. These ICs formulate necessary but not sufficient relations between the metamodels/models involved in each coupling process step. The ICs are formulated explicitly for specification-based information flow source code analyses addressed in this article. Necessary but not sufficient means that these conditions must hold to establish a coupling but the coupling is not guaranteed when they hold.

Reference Metamodels: Defining the ICs for specific analyses, their metamodels or security characteristics would restrict our coupling approach to a small set of analyses. Still, contribution **C2** also aims to achieve independence of specific metamodels, analyses, and security characteristics by specifying the ICs based on reference metamodels. The reference metamodels, presented in Section 6, define abstract syntactic elements to which elements in the input and output metamodels of the architectural analyses and source code analyses must conform. This abstraction from specific metamodels of the analyses provides knowledge that can be reused to couple other pairs of analyses. A black-box coupling also requires clear semantics of the elements in the models of the analyses to create valid transformations. Thus, we discuss semantics in the reference metamodels and ICs in Sections 6 and 7 respectively. As a result, the reference metamodels form interfaces for the analysis coupling because they are the target of the ICs. These interfaces are necessary for the black-box coupling [48]. Therefore, our coupling approach is only applicable to analyses for which the metamodels in the coupling conform to the reference metamodels.

4.1 Novelty

To the best of our knowledge, our coupling approach is novel by combining two aspects: (1) we propose a coupling process

that extracts values of security characteristics obtained with source code information flow analyses and integrates them into an architectural analysis. (2) With the reference metamodels and the ICs incorporated in our coupling approach, we provide reusable knowledge with defined interfaces to establish a coupling independent of specific analyses, their metamodels and security characteristics. This independence enables the application of our coupling approach to different architectural analyses with metamodels and models comprising different security characteristics of data to investigate various information-flow related security properties. In addition, our coupling approach is then applicable to existing information flow analyses with varying degrees of automation and quality properties, such as performance or accuracy. Our discussion of related work in Section 10 shows that related work does not cover these aspects.

4.2 Black-box Coupling

We decided for a black-box coupling because it establishes the coupling by using transformations between the input and output models of the analyses without combining their metamodels or introducing additional coupling-specific functionality in the tools. By way of contrast, if we designed our coupling approach as grey-box coupling (integrating coupling framework functionality and coordinating internal analysis steps) or as white-box coupling (merging metamodels), then the coupled architectural analysis would be executed only upon the availability of information from the source code analysis, i.e., when the values of the security characteristic of data resulting from an implementation of our newly designed architecture were available. However, it still would be necessary, for purposes of software evolution, to apply an architectural analysis in isolation (as discussed in Section 1). Thus, grey-box or white-box coupling requires maintaining two versions of the architectural analysis: one for the isolated analysis and one for the coupled analysis.

Hence, we see the following benefits of black-box coupling instead. 1) The black-box coupling approach does not require implementing the functionality of a coupling framework into the analysis tools (see Section 2.4). The black-box coupling only requires the execution of the transformations and isolated analyses to establish the coupling. Therefore, analysis tools can be freely used in isolation and by other coupling approaches. This benefit is especially relevant, as architectural analyses must be applicable in isolation, even in iterative scenarios when some implementation is available (see Section 1). 2) The exchangeability of the metamodels for different analysis tools is maintained, as an analysis tool does not encounter elements introduced only for the coupling. This contrasts with, for example, the Secure Link annotation in the source code used in GRaViTY [44]. 3) As long as the input and output metamodels are stable, the applied analysis techniques and analysis tools can be freely modified or exchanged. This capability increases the maintainability of the analysis tools used in the coupling. Analysis tools, for instance, can be modified to use other programming languages to increase analysis efficiency as illustrated by Boltz et al. [63]. 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. It also requires updating analysis tooling when the coupling functionality or the API of the applied coupling framework changes.

White-box coupling requires merging the (meta-)models of the analyses and creating new analysis tools to realize the coupled analysis. Thus, the previously mentioned benefits of our approach are also valid regarding white-box coupling. Still, the black-box coupling also has drawbacks, which are the topic of the discussion of limitations in Section 11.

4.3 Roles in the Coupling

Roles in our coupling process allow the separation of concerns in analysis development by assigning defined responsibilities and necessary competencies to actors. To handle complexity in the development of coupled analyses we extend the proposed roles by Koch [64] for creating single modular model-based analyses using analysis components. Our coupling approach adds to this by defining an additional role responsible for connecting single analyses to a coupled analysis. We want to highlight that an actor can have multiple roles. An actor creating a single software analysis may also be the one who establishes its coupling.

The *analysis component developer* of Koch [64] develops and maintains analysis components that encapsulate reusable functionalities for performing analyses. This role is not relevant in our coupling approach because it couples entire analyses rather than analysis components.

The *analysis architect* of Koch [64] assembles analysis components into a single analysis, which can be used in the coupling. In our coupling approach, *analysis architects* develop the architectural analysis and source code analysis independently of each other. Usually, these are two different teams without any overlap. The *analysis architect* has a white-box view of the single analysis, i.e., he or she can manipulate the analysis functionality and create the metamodels for the input and output of the analyses.

The new role of *analysis coupling expert* is responsible for the coupling between an architectural analysis and a source code analysis to a *coupled analysis* for a given *coupling scenario* using our coupling approach. For this, the *coupling expert* interacts with the *analysis architects* to first determine whether the source code analysis can analyze non-compliance regarding the security characteristic in the architectural analysis defined in a *coupling scenario*. Afterward, the *coupling expert* has to create the transformations defined in our coupling process. The *coupling expert* then has to determine correct correspondences and transformations between the input and output metamodels and models of the analyses. For this task, the *coupling expert* interacts with the *analysis architects* to ensure the correctness of the correspondences and the transformations. We guide this interaction with our ICs described later in Section 7. Thus, in contrast to the *analysis architect*, the *coupling expert* only needs to focus on the capabilities of the analyses and their input and output metamodels and models due to our black-box coupling. The *coupling expert* is also responsible for realizing the steps in our coupling process manually or by embedding them into a coupling tool that executes them.

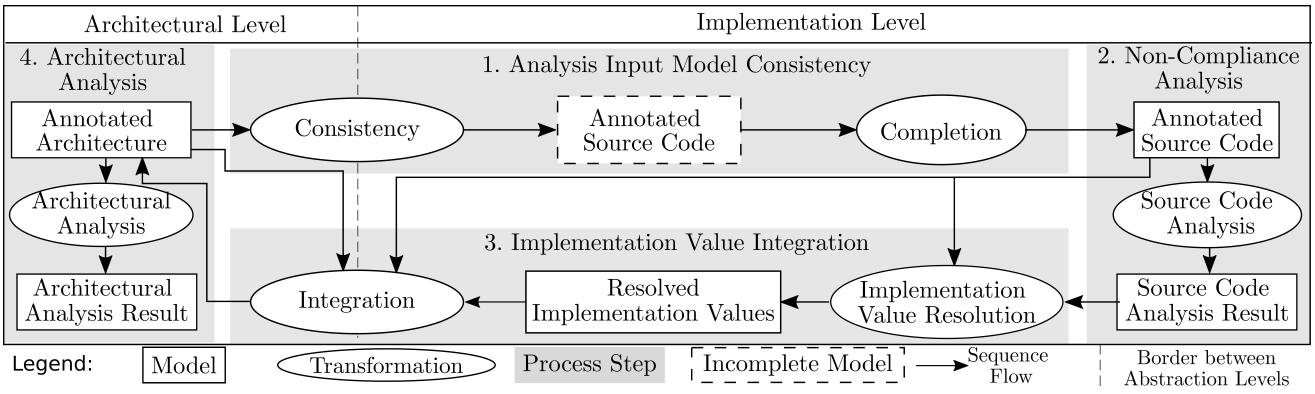

Fig. 3. Overview of the process steps in the proposed coupling approach with the comprised models and transformations

5 Coupling Process

Our coupling approach comprises the coupling process illustrated in Fig. 3 for realizing the coupling (**C1**). In a black-box coupling like ours (motivated in Section 4.2), the models are the primary artifacts; therefore, we begin by describing all models involved in our coupling process (Section 5.1). There are four process steps realized by transformations and their input and output models.

Analyses Input Model Consistency *(Section 5.2).* This step creates an analyzable input model of the source code analysis which is consistent with the input model of the architectural analysis.

Non-Compliance Analysis *(Section 5.3).* This step performs the source code analysis to detect non-compliance if it exists between the values of security characteristics of data resulting from the implementation and those specified in the input model of the architectural analysis.

Implementation Value Integration *(Section 5.4).* This step enriches the input model of the architectural analysis with the values of the security characteristics of data resulting from the implementation by modifying the values in the architectural specification.

Architectural Analysis *(Section 5.5).* This step executes the architectural analysis with the input model of the architectural analysis that has been enriched in the *Implementation Value Integration* step.

5.1 Models in the Coupling Process

The input and output models of the analyses are the main artifacts used in our coupling process. As both analyses use specifications of security characteristics, we call the input model of the architectural analyses *Annotated Architecture* and the input model of the source code analysis *Annotated Source Code*. We call the output of the architectural analysis *Architectural Analysis Result* and the output of the source code analysis *Source Code Analysis Result*. In this article, we focus on the *Source Code Analysis Result* providing counter-examples representing detected information flows that violate the given security policy and specification. The security characteristics resulting from the implementation must be obtained by interpreting the *Source Code Analysis Result*. When using a *Source Code Analysis Result* comprising the source and the sink of an information flow as in the running example, the data in the sink also contains data of the security characteristic of data of the source. We introduce the *Resolved Implementation Values Model* as a uniform abstraction of this interpretation, capturing the values of security characteristic resulting from the implementation. This uniform abstraction allows *coupling experts* to focus on creating the connection between the information about the values of security characteristics resulting from the implementation and the *Annotated Architecture* rather than on interpreting the *Source Code Analysis Result*.

5.2 Process Step: Analyses Input Model Consistency

The *Analyses Input Model Consistency* is the first process step in the coupling. It is responsible for providing an *Annotated Source Code* that can be analyzed with the source code analysis. In this process, the *Analyses Input Model Consistency* enforces that the system and specifications captured in the *Annotated Architecture* are consistently represented in the *Annotated Source Code*, especially regarding the security characteristic in the coupling scenario. For this purpose, the *Analyses Input Model Consistency* uses the *Annotated Architecture* as input and provides the *Annotated Source Code* as output. This consistent representation is a prerequisite to detect non-compliance between values of security characteristics of data specified in the *Annotated Architecture* and the values resulting from the implementation, as the specifications represent the same information. In the *Analyses Input Model Consistency* step, the transformations *Consistency* and *Completion* are applied.

5.2.1 Consistency Transformation

The *Consistency* transformation ensures (a) the consistent representation of the system in the *Annotated Architecture* and *Annotated Source Code* based on correspondences between their elements and (b) the consistent annotations of the values specified for security characteristic in the coupling scenario. This transformation is created by the *coupling expert* to connect the *Annotated Architecture* and the *Annotated 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 *Annotated Source Code*. These correspondences are then used to define transformations that modify elements in the *Annotated Source Code* based on the

Annotated Architecture. In our running example, transformations generate the method header and an empty body of *buy* from corresponding provided services of components and the values of the confidentiality levels *high*, *medium* and *low*. Please note that also, annotations for parameters can be generated, such as *high* for the parameter *userData* in the service *buy* or *low* from the annotations of the parameter *userData* in the service *store*. We discuss the generation of annotations when discussing efforts in Section 8.

In general, not all elements of the *Annotated Source Code* have corresponding elements in the *Annotated Architecture*, and vice versa, because of the abstraction gap between the system representations and differences in the specifications. For example, in our running example, the field *privilegeSum* and, by extension, the annotation of the value *medium* in the *Annotated Source Code*, are not represented in the component-based *Annotated Architecture*. We account for this by the *Consistency* transformation providing an *Incomplete Annotated Source Code*. Consequently, the *Incomplete Annotated Source Code* only captures elements for which correspondences exist between the *Annotated Architecture* and *Annotated Source Code*.

5.2.2 Completion Transformation

Software engineers must complete this *Incomplete Annotated Source Code* to the final system, which is performed in the *Completion* transformation. Examples of this task are completing method implementations or adding fields to classes. Other tasks, for example, include providing annotations not created in the *Consistency* transformation or the security policy if the architectural analysis captures it implicitly. Our running example highlights the importance of the *Completion* transformation by the illegal information flow from *privilegeSum* to *userData*. This information flow can only be detected and utilized if the method implementations, the field *privilegeSum* and its annotation are completed.

5.3 Process Step: Non-Compliance Analysis

The *Non-Compliance Analysis* step performs the *Source Code Analysis* transformation to identify non-compliance between the values of the security characteristic of the coupling scenario specified in the architectural model and those resulting from the implementation. The *Source Code Analysis* transformation uses the *Annotated Source Code* as input, executes the source code analysis and provides the *Source Code Analysis Result* as output.

The source code analysis inspects the implementation for information flows that are not allowed due to the specified values of security characteristics of data and a given security policy. These unallowed flows are reported as counter-examples in the *Source Code Analysis Result*. In the running example, two illegal flows to the parameter *userData* in the method *store* are reported (partial model (d) in Fig. 1): One from the field *priveledgeSum* with the confidentiality level *medium* and one from the parameter *userData* with the confidentiality level *high* in the method *buy*. This is because both violate the security policy (partial model (c) in Fig. 1) that only data with lower confidentiality must flow to data with higher confidentiality.

When regarded in isolation, these counter-examples describe non-compliance between the specifications in the Annotated Source Code and the implementation. However, the values of the security characteristics are consistent for corresponding system elements in the *Annotated Architecture* and the *Annotated Source Code* due to the *Analyses Input Model Consistency* step. Consequently, the counter-examples also describe non-compliance between the implementation and the specification in the *Annotated Architecture*.

5.4 Process Step: Implementation Value Integration

The *Implementation Value Integration* step integrates the values of security characteristics resulting from the implementation, uncovered by the application of the source code analysis in the *Non-Compliance Analysis* step, into the *Annotated Architecture*. This integration enables the usage of the values resulting from the implementation in the architectural analysis. For this purpose, the *Implementation Value Integration* uses the *Source Code Analysis Result*, the *Annotated Source Code*, and the *Annotated Architecture* as input and provides a modified version of the *Annotated Architecture* as output. The values resulting from the implementation must be extracted from the counter-examples of the *Source Code Analysis Result* before performing an integration. The *analysis architect* knows the meaning of the counter-examples and their relation to the implementation, which is required for this extraction. In contrast, integrating the values resulting from the implementation into the *Annotated Architecture* requires bridging the gap between the abstractions, which is the responsibility of the *coupling expert*. We account for these different responsibilities by separating the *Implementation Value Integration* step into the transformations *Implementation Value Resolution* and *Integration*.

5.4.1 Implementation Value Resolution Transformation

The *Implementation Value Resolution* transformation is defined by the *analysis architect* as they know how the reported information flows influence the security characteristics of data of the implementation elements. The *Implementation Value Resolution* uses the *Source Code Analysis Result* and *Annotated Source Code* as input, providing the *Resolved Implementation Values Model* as output. The *Implementation Value Resolution* interprets the *Source Code Analysis Result* and calculates the values of the security characteristic of data resulting from the implementation for implementation elements. This interpretation requires considering several counter-examples. In our running example, the source code analysis provides two counter-examples for the parameter *userData* in the store method: the flow from the medium classified field *privilageSum* (11) and the flow from the high classified parameter *userData* in the *buy* method (12). For strictly ordered *lattices* as in the source code analysis security policy (partial model (c) in Fig. 1), the *Implementation Value Resolution* can be realized as follows: We collect all values of confidentiality levels from the sources of the flows reported by the source code analysis. We select the value resulting from the implementation by following the lattice to the value to which no flow is allowed from any other value. In our running example, this results in the level *high* for the parameter *userData* in the *store* service. The rationale behind this approach is that if a violation arises due to *medium*, it also arises because of *high*. Considering 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. This calculation is not necessary for our running example as the *Source Code Analysis Result* already contains all values available in the *Annotated Source Code* (*low*, *medium* and *high*). Still, we address this by using the *Annotated Source Code* as additional input for the *Implementation Value Resolution* transformation to verify if a calculated new value exists in the specification of the implementation.

5.4.2 Integration Transformation

The *Integration* transformation uses the *Resolved Implementation Values Model* and *Annotated Architecture* as input and provides a modified version of the *Annotated Architecture* as output. This transformation is created by the *coupling expert* as it bridges the gap between the implementation and the architecture. The *Integration* transformation performs two steps: First, the annotations in the *Annotated Architecture* are resolved, which must be modified based on the information captured in the *Resolved Implementation Values Model*. Secondly, the annotated values of the corresponding security characteristic are modified based on the values represented in the *Resolved Implementation Values Model*. The *Annotated Architecture* is considered the prescriptive system description (see Section 1). Changing the original *Annotated Architecture* in our coupling process would result in losing information like design decisions or stakeholder requirements. This is why the coupling approach creates a modified copy of the *Annotated Architecture* for detecting vulnerabilities arising from said non-compliance to avoid this loss of information.

5.5 Process Step: Architectural Analysis

In the *Architectural Analysis* step, the architectural analysis is executed on the result of the *Annotated Architecture* resulting from the *Implementation Value Integration* step. Thus, *Architectural Analysis* step allows to detect the effect of identified non-conformances between the values of security characteristics of data specified in the architectural model and those resulting from the implementation. This step is necessary to determine whether the architectural analysis reports new vulnerabilities based on the values modified by the coupling. This is the case if modified values violate the applied security policy. However, as illustrated in the running example, not every modification of values of security characteristics of data result in new vulnerabilities. We do not detail the *Architectural Analysis* step any further in the remainder of this article as it does not affect the coupling.

6 Reference Metamodel for Metamodels of Analyses in the Coupling

We abstract from specific analyses by describing reference metamodel for the metamodels of the analyses involved in the coupling as part of contribution **C2**. The reference metamodels contain metaclasses and references between the metaclasses relevant to our coupling approach expected to be represented in the metamodels of the analyses. An analysis can be used in the coupling if it conforms to the reference metamodels. We define this conformance later in Section 6.4.

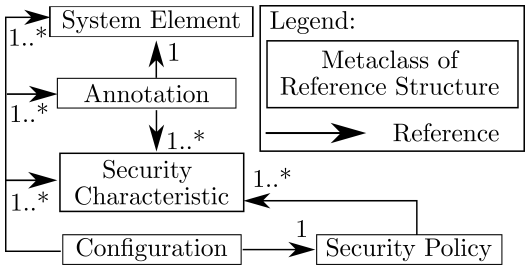

Fig. 4. The syntactic elements of the reference metamodel for input metamodels of the architectural analysis and code analysis

We detail a reference metamodel for the *Annotated Architecture* and the *Annotated Source Code* in Section 6.1 and for the *Source Code Analysis Result* in Section 6.2. In addition, we define a reference metamodel for the *Resolved Implementation Values Model* in Section 6.3. We do not provide a reference metamodel for the *Architectural Analysis Result* because it is not involved in the actual coupling.

6.1 Reference Metamodel: Input Metamodels

The reference metamodel for the *Annotated Architecture* and *Annotated Source Code* is presented in Fig. 4. We structure the reference metamodel in *System Elements*, describing the system, *Security Specification Elements* for the representation of the specification, and *Configuration Elements* used for configuration of the analysis. In this section we clarify the semantics of the elements in the reference metamodels as this must be clear to create valid couplings [47].

6.1.1 System Elements

A *system element* is a element to describe the system in a design language or programming language. A system element represents structural information, such as software components, or behavioral information, such as expressions in a method. In the running example, *System Elements* are, e.g., parameters or resources in the *Annotated Architecture* or classes, methods and fields in the *Annotated Source Code*.

6.1.2 Security Specification Elements

The security specification comprises the metaclasses *Security Characteristic* and *Annotation* that are used to describe security-relevant properties of *System Elements*.

Security Characteristic: A security characteristic describes security-related properties of *System Elements* (see Section 2.2). The *Type* is captured in a specific metamodel by the metaclass of the security characteristic itself, e.g., *confidententiality level* is the *Type* of the security characteristic in the running example. In the running example, the values *high* and *low* are instances of *confidentiality level*. The semantics of a syntactic element representing a security characteristic can vary, depending either on the described *Type* or the values in case they are specifiable by the analysis user. A metamodel, for instance, can define a metaclass representing confidentiality levels, similar to Tuma et al. [19]. However, another metamodel may provide a more general metaclass, such as *Level* in *JOANA*, allowing the specification of, amongst others, confidentiality levels or roles as required to describe and analyze the system.

Annotation: *Annotations* enable the specification of *Security Characteristics* for *System Elements*. For this, an *Annotation* references one or more *Security Characteristic* and one *System Element*. 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*. *JOANA* [23], for instance, provides the *Annotations Source* and *Sink* both specifiying the the *security characteristic of data level*. When specifying the same parameter with the value *low* and the value *high* in a *Source* and *Sink* respectively at the same time, it would be unclear whether the parameter is expected to contain *high* or *low* data.

6.1.3 Analysis Configuration Elements

The analysis configuration metaclasses comprise the *Security Policy* and *Configuration*. They define the system and specification used in performing the analysis and when the analysis must report a vulnerability.

Security Policy: Analyses detect vulnerabilities based on a *Security Policy*. The *Security Policy* references one or more *Security Characteristics* for describing the allowed or unallowed relations between them. Thus, possible values of *Security Characteristics* occurring in the *Source Code Analysis Result* are based on the values described in the *Security Policy*.

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 *Implementation Value Integration*. An analysis uses a given *Configuration* to identify vulnerabilities. A *Configuration* defines information relevant to an execution of an analysis (i.e., the system to be analyzed, the available values of the *Security Characteristics*, their *Annotations* to *System Elements*, and the applied *Security Policy*). In our running example, the configuration in *JOANA* [23] defines the source code to be analyzed, the available security level, the applied annotations, and the applied lattice. The *Configuration* metaclass references the *Security Characteristic* to define which values of security characteristics are used in the analysis. In addition, a *Configuration* references one or more *Annotations* used in the analysis. Allowing different annotations enables what-if analysis or supporting constraints in analysis algorithms. *JOANA*, for instance, does not allow the annotation of a single *System Element* as *Source* and *Sink* in the same configuration. Please also note that the requirement about all *Annotations* of the same semantic domain annotating the same values of the *Security Characteristics* only holds in a *single Configuration*. The *Configuration* references a *Security Policy* because the analysis results represent policy violations.

Several analyses such as [6], [12], [18], [19], [36] are configured by the input of the analysis without a dedicated configuration metaclass. In other analyses such as [23], [65], there may be a configuration metaclass identifying the *Annotations*, *Security Characteristics* and *Policies*, but the system to be analyzed is provided as dedicated input for the analysis. We account for the configuration without a dedicated metaclass or a mixture of both by defining the totality of all input models for the analyses as *Configuration*. We call a *Configuration explicit* if a dedicated metaclass exists and *implicit* otherwise.

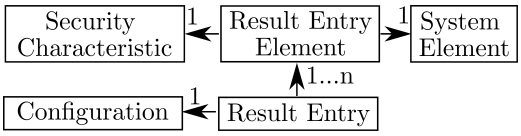

Fig. 5. The syntactic elements of the reference metamodel for the output model of the code analysis. Please see Fig. 4 for the legend.

6.2 Reference Metamodel: Output Metamodels

The reference metamodel for the *Source Code Analysis Result*, illustrated in Fig. 5, defines metaclasses for counter-examples expressing the violation of the given specification and security policy rather than quantitative values or binary yes-or-no statements. A common structure for the *Source Code Analysis Result* in information flow analysis is to represent the origin (source) and destination (sink) of an information flow that lead to a vulnerability (e.g., in [12], [23], [65], [66]). In the running example, an illegal flow is represented by providing the *privilegeSum* with the confidentiality level *medium* as source and *userData* with the confidentiality level *low* as sink.

The metaclass *Result Entry* in our reference metamodel represents a single counter-example. For structuring the counter-example, the *Result Entry* comprises a set of *Result Entry Elements*. 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 Section 6.1.2), a combination of *System Element*, *Security Characteristic*, and *Configuration* suffices to resolve annotations in the input model. Consequently, a *Result Entry Element* references a *System Element* and a *Security Characteristic*. In addition, the *Result Entry* references a *Configuration* to relate the found counter-examples to the input and to enable the resolution of the *Annotations*. These metaclasses also exist in the reference metamodel of input metamodels because the *Source Code Analysis Result* is an abstraction of the *Annotated Source Code* [54]. 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).

6.3 Reference Metamodel: Resolved Implementation Values Model

The reference metamodel for the *Resolved Implementation Values Model* (see Section 5.1) is similar to the one for the *Source Code Analysis Result* but abstracts the counter-examples and the interpretation performed by the *analysis architect* in the *Implementation Value Resolution* transformation. The reference metamodel for the *Resolved Implementation Values Model* comprises the metaclass *Resolved Implementation Value*. This metaclass references a *Security Characteristic*, a *System Element*, and a *Configuration*, as this information suffices to identify the annotations and security characteristic to be modified in the *Implementation Value Integration*.

6.4 Conformance of Metamodels of Analyses to the Reference Metamodels

An analysis can only be used in the coupling if its input and output metamodels conform to our reference metamodels. Similarly to Konersmann et al. [61], we comprehend the reference metamodels as underspecified, i.e., the specific metamodels of the analyses can comprise metaclasses and references not captured in the reference metamodels. Consequently, 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 according to Definition 4.

Definition 4 (Conformance of a Metaclass of Metamodels to a Metaclass of the Reference Metamodel).
A metaclass $m \in M$ in the input or output metamodel of an analysis conforms to a metaclass $r \in R$ in the reference metamodel if a valid transformation can be defined. 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 $r' \in R$, the metaclass m must provide a (possibly transitive) reference to another metaclass $m' \in M$ conforming to r'. We define a reference as transitive if metaclass m' is reachable through a sequence of references starting from m.

An annotation specifying the confidentiality levels in our running example has the semantics of the *Annotation* in the reference metamodel. In addition, the annotation references a *parameter* conforming to a *System Element*, and it references the confidentiality level conforming to *Security Characteristic*. Therefore, the annotation conforms to the *Annotation* of the reference metamodel according to Definition 4.

Similar definitions of conformance can be found in related domains. For example, conformance is defined between two models of the same metamodel [6], it is defined between reference models and specific models created with the same metamodel [61], and it is defined between specific components of a specific architecture and an abstract component of a reference architecture [67].

7 Integration Conditions

We now define the Integration Conditions (ICs) for each process step as part of contribution **C2**. For realizing the coupling in the *Implementation Value Integration*, the annotations annotating the security characteristic of data of the coupling scenario in the *Annotated Architecture* must be retrieved and the specified values must be replaced with those resulting from the implementation. Consequently, we have to ensure that the *Implementation Value Integration* receives a *Resolved Implementation Values 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 Sections 6.1 and 6.2, we can unambiguously identify an *Annotation* by a *Security Characteristic*, *System Element*, and a *Configuration*. Our ICs ensure resolution capabilities between the *Resolved Implementation Values Model* and *Annotated Architecture* for these elements to achieve (1) and (2). A *coupling expert* can also use the ICs to elaborate correspondences for the transformations in the coupling process.

Formulating the ICs on the metamodel level does not suffice, as the validity of a resolution may also depend on the specific models due to *model-level semantic refinements* (see Section 2.5). Thus, we specify the ICs to hold on the metamodel and model level. Due to space limitations, we describe the ICs on the metamodel level and omit their description on the model level in this article. Nonetheless, we provide here one *example for the first Integration Condition (IC1)* on the model level. Beyond that, we provide all ICs on both the metamodel and the model level in Supplementary Material [68].

We classify the ICs in *correspondence conditions* and *constraint conditions*, respectively abbreviated with T and R. *Correspondence conditions* induce 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. Some correspondence conditions describe correspondences between different metaclasses of the reference metamodels to increase conciseness in this article. We refine these conditions into more fine-grained ICs in Supplementary Material [68] if possible. *Constraint conditions* constrain 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.

Finally, the coupling is only correct if the correspondences between the metaclasses in the metamodels of the analyses are valid on the syntactic *and* semantic level [48]. Therefore, we expect the correspondences induced by the correspondence conditions to be valid on the syntactic and semantic level, as discussed in the next section.

7.1 Validity of Correspondences

The correspondences between the metamodels and models of the analyses for the coupling, established by the *coupling expert*, must be valid. Considering the validity of correspondences in the ICs only on the syntax does not suffice [47], [54], as illustrated by the following example: A *correspondence condition* induces a correspondence between an annotated *System Element* in the *Annotated Architecture* and an annotated *System Element* in the *Annotated Source Code*. According to this condition, a correspondence between a parameter and a field in the running example would be valid without considering semantics, as both are annotated *System Elements* in their respective models. In this example, the *coupling expert* could determine the invalidity of the correspondence through the names of the syntactic elements. However, a correspondence between the syntactic element *Security Characteristic of Data* in the *Annotated Architecture* and the syntactic element *Confidentiality Level* in the *Annotated Source Code* in our running example is not that obvious.

This is especially true if a *model-level semantic refinement* exists. Therefore, we require that the *syntactic elements* in a *correspondence* induced by a *correspondence condition* have a *semantic mapping* to the same *semantic element*.

7.2 Integration Conditions: Analyses Input Model Consistency

The *Annotated Source Code* connects the *Source Code Analysis Result* and the *Resolved Implementation Values 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*.

For this, we require with **IC1** that the security characteristic in the coupling scenario in the *Annotated Architecture* must be resolvable from the *Annotated Source Code*. As the *Resolved Implementation Values 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): A *Security Characteristic* must exist in the *Annotated Source Code* which corresponds to the *Security Characteristic* of the coupling scenario in the *Annotated Architecture*.

> **IC1**(T)(Model Level Example): An instance of a *Security Characteristic* must exist in the *Annotated Source Code* which corresponds to an instance of the *Security Characteristic* of the coupling scenario in the *Annotated Architecture*.

For resolving annotations, asides from the security characteristics we also require the annotated system elements and the configurations to be resolvable. Therefore, we require with **IC2** resolution capabilities between the system elements and configuration involved in the annotation of security characteristics in the coupling scenario.

> **IC2**(T): There must exist 1) *System Elements* of the *Annotated Source Code* which correspond to *System Elements* of the *Annotated Architecture* that are annotated with the *Security Characteristic* of the coupling scenario by at least one *Annotation* and 2) *Configurations* of the *Annotated Source Code* which correspond to *Configurations* in the *Annotated Architecture* that uses at least one *Annotation*, annotating the *Security Characteristic* in the coupling scenario.

In the running example, this IC is fulfilled as *Parameters* of methods in the implementation can be resolved to *Parameters* in services, e.g., the *userData* in the *store* method in the implementation can be resolved to the *userData* in the *store service* architecture. We assume for the running example for both analyses implicit configurations for simplicity. Thus, the configurations of the architectural analysis can be resolved through the applied configuration of the source code analysis.

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** introduces an implicit correspondence between the annotations in the *Annotated Architecture* and in the *Annotated Source Code* based on the assumption from Section 6.2 that an annotation can be identified by the system element, security characteristics and configuration. In addition, while 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.

> **IC3**(C): There must exist an *Annotation* in the *Annotated Source Code* that annotates a *Security Characteristic* affected by **IC1** to a *System Element* affected by **IC2** and is referenced by a *Configuration* also affected by **IC2**.

In the running example, **IC3** holds because the *Annotated Source Code* comprises an annotation, which annotates the *confidentiality level* like *high* (**IC1**) to *Parameters* such as *userData* (**IC2**) while using an implicit configuration (**IC2**).

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 architect* have to define a security policy for the source code analysis. We guide the manual creation and the transformation by **IC4**.

> **IC4**(C): *Security Policies* of the source code analysis must use the *Security Characteristics* affected by **IC1** of their *Configuration*.

IC4 avoids encounters, later in the process, with security characteristics of data 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*.

7.3 Integration Conditions: Source Code Analysis

In the *Implementation Value Integration* transformation, the values of security characteristics resulting from the implementation in the *Resolved Implementation Values 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**.

We do not specify in the reference metamodels which security characteristics or values must be part of the counterexample. Thus, we specify **IC5** to avoid the sole availability of system elements and security characteristics that are not

resolvable to the *Annotated Architecture*, e.g., fields in the running example.

> **IC5**(T): There must exist *Security Characteristics* and *System Elements* in the *Source Code Analysis Result* corresponding to *Security Characteristics* and *System Elements* of the *Annotated Source Code* that are affected by **IC1** and **IC2** respectively.

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. However, **IC6** still allows counter-examples for flows from or to *Fields*.

> **IC6**(C): There must exist a *Result Entry Element* in the *Source Code Analysis Result* containing a *System Element* and a *Security Characteristic*, both affected by **IC5**.

IC5 and **IC6** both hold in the running example as the *Result Entry* contains the *Result Entry Element* containing the *userData* of the *store* method and the confidentiality level *low*, which can both 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): It must exist *Configurations* in the *Source Code Analysis Result* corresponding to *Configurations* of the *Annotated Source Code* affected by **IC2**.

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.

7.4 Integration Conditions: Implementation Value Integration

All elements in the *Resulting Values Entry* of the *Resolved Implementation Values 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*. **IC8** establishes this capability in the *Resolved Implementation Values Model* by utilizing the elements in the *Source Code Analysis Result* corresponding to **IC2**.

> **IC8**(C): In all *ResolvedImplementationValue* in the *Resolved Implementation Values Model*, the contained *System Elements* and *Configurations* must correspond to *System Elements* and *Configurations* in the *Source Code Analysis Result*, which are affected by **IC5** and **IC7**, respectively.

IC8 can be established in our running example by providing an *Resolved Implementation Values Model* which only contains the system element *Parameter*, as these are the only annotated elements in the *Annotated Architecture* that are affected by **IC2**. An *Resolved Implementation Values Model* which only contains fields would not be applicable in the *Implementation Value Integration* of our running example as fields are not represented in the component-based *Annotated Architecture*. **IC8** can also be established in our running example when the *Resolved Implementation Values 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*. Therefore, a *analysis architect* can construct the *Resolved Implementation Values Model* and *Implementation Value Resolution* to contain security characteristics that cannot be resolved to the *Annotated Architecture*, e.g., locations in the prior example. In addition, when the interpretation in the *Implementation Value Resolution* calculates values unaffected by **IC1**, a corresponding element in the *Annotated Architecture* cannot be resolved. An *analysis architect* unaware of different semantics of the values of security characteristic of data, for instance, could create an *Implementation Value Resolution* that calculates the value *LowMediumHigh* for the parameter *userData*. However, this value cannot be mapped to the *Annotated Architecture* as it contains the value 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**. We formulate **IC9** to ensure that the security characteristics and their calculated values captured in the *Resolved Implementation Values Model* can be resolved to the *Annotated Architecture*.

> **IC9**(T): All *ResolvedImplementationValue* in the *Resolved Implementation Values Model* have to contain a *Security Characteristics* corresponding to *Security Characteristics* in the *Annotated Source Code* affected by **IC1**.

Because **IC8** and **IC9** are defined for all elements in the *Resolved Implementation Values 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 *Resolved Implementation Values 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 *Resolved Implementation Values Model* does not consider it for analysis. Thus, we constrain the security characteristics in an *Resolved Implementation Values Model* with **IC10**.

> **IC10**(C): The *Security Characteristics* in the *Annotated Source Code* corresponding to the *Security Characteristic* in a *Resolved Implementation Value* in the *Resolved Implementation Values Model* affected by **IC9** must be referenced in a *Configuration* in the *Annotated Source Code* corresponding to the *Configuration* affected by **IC8** in the same *Resolved Implementation Value*.

IC10 would hold in our running example, when the *Resolved Implementation Values Model* only comprises the values *high*, *medium* and *low* from the security characteristic confidentiality level as the implicit configuration in the *Annotated Source Code* comprises corresponding values.

```
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 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*.

8 Efforts in the Coupling Process

We now discuss efforts that arise when realizing the 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.

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

The *Consistency* transformation in the *Analyses Input Model Consistency* step is expected to be fully automated as all manual effort is deliberately located in the *Completion* step. This automation can be achieved using consistency preservation approaches such as Vitruvius [69] or generative approaches like [53], [70]. A significant effort in the coupling results from the definition of the annotation, i.e., the specification of the security characteristics. In Section 6, 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 consistency preservation for elements captured by **IC1** to **IC3** in previous work [71]. For our running example, applying the example code in Listing 1 would result, amongst others, in the following transformations: 1) Mapping the implicit configuration $config_A$ of the architectural analysis to a configuration $config_C$ of the source code analysis (**IC2**). 2) Mapping the *confidentiality levels high*, *medium* (not shown in Fig. 1 for the architecture), and *low* comprised 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*. 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**). This results in a one-time effort to create the *Consistency* transformation for all source code elements corresponding to the architectural analysis's input (meta)model. More fine-granular effort in creating the *Consistency* transformation can arise if the transformations have to be parameterized based on different semantics on the model level because of a *model-level semantic refinement* (see Section 2.5). While this results in additional transformation rules, they have to be created once for the *Consistency* transformation for a pair of analyses. We provide examples for such transformations in our replication package [72] for our evaluation in this article (see Section 9).

The *Completion* transformation in the *Analyses Input Model Consistency* step is expected to be performed manually as its purpose is to complete the *Annotated Source Code* for all elements without correspondence to the *Annotated Architecture*, which are used in the *Consistency* transformation. The manual effort in the *Completion* transformation depends on the degree of transformation rules to generate elements in the *Annotated Source Code* that can be defined by the *coupling expert* in the *Consistency* transformation. One of the major efforts in the *Completion* transformation is the definition of annotations, which cannot be generated in the *Consistency* transformation if they are not covered by our ICs. Annotations for fields in the running example are an example of annotations not covered by our ICs. The definition of such annotations has to be performed manually in every application of the coupling approach. Polikarpova et al. [73] explain that software engineers can write simple specifications competently when expressed in a similar syntax. In the context of this article, the specifications comprise the annotations of security characteristics of data to system elements, which we assume to be simple. Still, as the capability depends on familiarity with the syntax and we aim to support arbitrary analyses, providing more exact estimations of effort is challenging.

As we expect an evolution scenario, only those parts of the implementation must be provided manually in *every 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. Using a generative approach for the *Consistency* transformation may require 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, for instance, by using consistency preservation approaches or by using tools to merge generated and existing source code like our prototypical implementation [74]. 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 *coupling expert* may be able to provide an automated generation of a lattice in the *Annotated Source Code* by exploiting knowledge about the implicit security policy used by the architectural analysis, which states that confidential data must not flow to public data. In this way, the security policy can be generated automatically in the *Consistency* transformation, rather than being manually provided in the *Completion* transformation. We provide examples for such generation in the replication package [72] for our evaluation in this article (see Section 9).

The effort in the *Source Code Analysis* step depends on the source code analysis applied. Fully automated analyses like *JOANA* [23] 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 [75] with automated execution of source code analyses in our evaluation described in Section 9. Analyses with manual intervention, such as the information flow analysis of Greiner [25], must be executed manually in each execution of the coupling process, resulting in a repeated effort.

The *Integration* and *Implementation Value Resolution* transformations are designed to be fully automated. In the *Implementation Value Resolution*, the one-time efforts involve the *coupling expert* and the *analysis architect* to determine the transformation between the *Source Code Analysis Result* and the *Resolved Implementation Values Model*. The effort for the *Implementation Value Resolution* again rises if different semantics have to be supported. However, creating these additional transformation rules again imposes a one-time effort for the *coupling expert* and the *analysis developer* for every addressed semantics. The *Integration* transformation imposes a one-time effort for the *coupling expert* to define how the information in the *Resolved Implementation Values Model* is integrated into the *Annotated Architecture*.

In summary, only the creation of the *Annotated Architecture* and the *Completion* transformation always comprise guaranteed manual efforts that must be performed in each execution of the coupling. 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. Still, the effort for this one-time creation depends on the experience of the *coupling expert* and the familiarity with the analyses to be coupled.

9 Evaluation

We now evaluate our two contributions, beginning at **C2** and then proceeding to **C1**. The rationale here is this: First, the relations between the reference metamodels and specific metamodels and also the relations between our ICs and successful couplings (**C2**) need to be evaluated. Secondly, only then can we evaluate whether we achieve the goal of our coupling approach to detect architectural vulnerabilities based on security characteristics of data resulting from the implementation when the coupling for architectural analyses and source code analyses is established with our coupling process (**C1**). In short, the reference metamodels and the ICs (detailed in Sections 6 and 7) guide the establishment of the coupling process (detailed in Section 5).

Our evaluation is guided by the application of the Goal-Question-Metric (GQM) approach [76], as presented in Section 9.1. In Section 9.2, we describe how we collect the metrics based on our study design. In Section 9.3, we present and discuss the results of the evaluation. In Section 9.4, we consider the threats to the validity of our evaluation.

9.1 Goals, Questions and Metrics

Our evaluation follows the Goal-Question-Metric (GQM) approach [76]. According to GQM, the evaluation is guided by first setting the evaluation goals, then asking the questions to determine whether we reach those goals, and lastly defining metrics to answer the questions. This approach allows for systematic structurization of the evaluation before creating the study design and executing the experiments.

To begin, we evaluate **C2** by setting the first goal:

- **G1** *Coverage of Reference Metamodels and Integration Conditions*.

We investigate whether (a) the metaclasses of the reference metamodels are covered by the input and output metamodels of the analyses and (b) whether all ICs are covered by the transformations in successful couplings. The reference metamodels contain the metaclasses and the references *between* the metaclasses that are needed to perform the coupling. Therefore, for the coupling to be applicable, the metaclasses of the reference metamodels must occur in the input and output metamodels of the analyses. We ask the question **Q1.1**:

- **Q1.1** Are all metaclasses and relations between these metaclasses in the reference metamodels *covered by* the input and output metamodels of the architectural analyses and source code analyses in the coupling which consider information flow-related security characteristics?

The reference metamodels build a foundation for our ICs. The ICs define *necessary but not sufficient* conditions for the relations between the metamodels (and their models) of the analyses. These conditions enable a coupling approach to

successfully integrate the values of security characteristics from the implementation into the architectural-analysis input model. Consequently, any condition that does not hold *despite successful establishment of the coupling* is hence unnecessary for the coupling. Therefore, we ask our second and third questions about the ICs and their coverage on the metamodel level (**Q1.2**) and on the model level (**Q1.3**):

- **Q1.2** Upon the successful establishment of coupling, are all ICs *covered by* the transformations in the coupling process between the *metamodels* of the architectural analysis and the source code analysis?
- **Q1.3** Upon the successful establishment of coupling, are all ICs *covered by* the transformations in the process between the *models* of the architectural analysis and the source code analysis?

Related work, such as [61], [67], [77], [78], also evaluates coverage similarly to the way we do here via our three questions, but instead of the term coverage, those works use terms like conformance [61], [67] or soundness [77]. The difference between related evaluations and ours is this: The related work evaluates coverage relations between specific artifacts by defining specific metrics, for example, by comparing taxonomies and sentences [77], by comparing reference architectures and specific architectures [67], by comparing whether a reference model is realized by specific models [61] or by comparing secured operations and measured API endpoints [78]. In contrast, here we evaluate coverage for different artifacts.

For a more comprehensive evaluation, we first generalize **Q1.1**, **Q1.2**, and **Q1.3** as well as the metrics of related work to the metric $Coverage$ in Eq. (1).

$$Coverage(A, COND) = \frac{\sum_{cond \in COND} hold(A, cond)}{|COND|} \quad (1)$$

with

$hold\colon A \times COND \rightarrow \{0, 1\}$

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 $hold$. We describe this specialization for **Q1.1**, **Q1.2**, and **Q1.3** in Section 9.2.2. Furthermore, we apply our specialization to a case study (described in Section 9.2), in order to answer **Q1.1**, **Q1.2**, and **Q1.3**.

Next, we evaluate **C1** by setting the second goal:

- **G2** *Accuracy regarding Architectural Vulnerabilities Arising from Non-Compliant Information Flows in the Implementation*.

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 non-compliant security characteristics of data resulting from information flows in the implementation.

To this end, we assume that the static source code analysis is accurate, i.e., that it detects vulnerabilities if they exist and that it does not raise false alarms. We admit that this assumption is unrealistic. However, it is appropriate for our evaluation as described in the following. While the accuracy of static information flow analyses is evaluated in the literature [11], [12], [79], the results show that perfect results are hard to achieve and that often over-approximations occur. 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, in fact, lead to false alarms in the architectural analysis as well. Also, existing information flows that are unallowed but also go undetected by the source code analyses cannot be used in the architectural analysis (see Section 11). Despite all this, we deliberately assume the accuracy of the source code analyses because we intend for our evaluation to (1) simulate a scenario where a *coupling expert* cannot influence the accuracy of a source code analysis, (2) define a ground truth without overfitting to the analysis capabilities, and lastly (3) allow investigating the impact of inaccuracies on the results of the architectural analysis.

Based on this assumption, we therefore ask question **Q2** about the accuracy of architectural analysis in the coupling:

- **Q2** How accurate is the coupled analysis in detecting architectural vulnerabilities arising from non-compliant information flows in the implementation?

In related work (e.g., [6], [12], [36], [44]), the metrics commonly used to evaluate accuracy in analyses are precision p and recall r. Precision p and recall r are calculated by considering true positives t_p, false positives f_p, and false negatives f_n in the expected results of the architectural analysis. We use these metrics to evaluate the accuracy enabled by the coupling.

$$p = \frac{t_p}{t_p + f_p} \qquad r = \frac{t_p}{t_p + f_n}$$

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 by using our case study (described in Section 9.2).

9.2 Study Design

We base our evaluation on a case study because, not only is this the preferred method of evaluation in related work (e.g., [6], [61], [67], [77]) but also because case study is one of the most applied methods for evaluating notations of models for analyses [80]. We use our case study to collect data on our metrics *Coverage* and *Accuracy* (detailed in Section 9.1). To that end, we define cases that comprise a *coupling scenario* and a *system* to be analyzed. Each system comprises: 1) an

architectural model of the system 2) the implementation of the system and 3) the specification of information flow-related security characteristics. Our case study comprises *sixteen cases*. We couple *two* architectural analyses with *two* source code analyses for *one* security characteristic of data, obtaining a total of *four* coupling scenarios. We then apply these *four scenarios* to *four* case study systems. Our selection of four analyses results in *four* metamodels for their input and output and also *two* metamodels for the *Resolved Implementation Values Model*. Executing our analysis coupling in all 16 cases results in a total of *88* models: *8 x 1* (i.e., the *Annotated Architecture*) and *16 x 5* (i.e., the *Annotated Source Code*, the *Source Code Analysis Result*, the *Resolved Implementation Values Model*, the modified version of the *Annotated Architecture*, and the *Architectural Analysis Result*).

For our case study, 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. We select case study systems with different semantics of the security characteristics of data. Two of our systems specify confidentiality levels, whereas the other two systems use specifications of roles.

To evaluate **G1** *Coverage of Reference Metamodels and Integration Conditions*, we require both input and output metamodels and the models used in the coupled analyses. We require these metamodels and models to be usable in the Eclipse Modeling Framework (EMF) [81], a popular open-source framework for model-driven development used in literature like [6], [18], [19], [21], [28], [35], [36], [37], [44], [62]. 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. For the input models of analyses, we either reuse them from related work or, where missing, we recreate them manually as described in this section and further discussed in our supplementary material [68]. To obtain the output model of an analysis, we apply source code analysis to the respective input model.

Another requirement for the evaluation of **G1** is that the correspondences be represented in correspondence models. For this, we ourselves define the necessary correspondence metamodels (for all pairs of analyses in the case study) and the metamodel of the *Resolved Implementation Values*. To capture correspondences with an implicit configuration (see Section 6.1.3), we introduce a metaclass in the correspondence metamodels that references all metamodels of the analysis (see Supplementary Material [68] for details).

Our metric *Coverage* must first be specialized so that we can use it to evaluate the coverage of the reference metamodels for answering (**Q1.1**) and the coverage of the integration conditions for answering (**Q1.2** and **Q1.3**) (see Section 9.1). We describe this specialization in Section 9.2.2.

To evaluate **G2** *Accuracy regarding Architectural Vulnerabilities Arising from Non-Compliant Information Flows in the Implementation* for a given coupling, we require that values of the security characteristics of data in the implementation affected by information flows must be non-compliant with the values specified in the *Annotated Architecture*. To collect our metrics *precision* and *recall* (see Section 9.1), we define the ground truth in Section 9.2.3 — specifically, we use injected information flows as our basis for estimating the reports of the architectural analysis that should occur after the coupling. Also, we describe our calculation of the true positives, false positives, and false negatives.

The coupling process is performed by applying our prototypical coupling framework [75], which executes the process steps of our coupling approach. All automated transformations embedded in the coupling framework are included in our replication package [72]. The metamodels and models of our case study are also included in our replication package [72]. We also provide further details about our cases in the Supplementary Material [68].

9.2.1 Selection of Cases

Here we discuss our selection of the analyses and systems in the case study.

9.2.1.1 **Selection of Analyses**: To select the architectural and source code analyses in the case study, we first classify these analyses by specific attributes, which we discuss in the following. It is our aim to use a broad set of representative analyses to evaluate our coupling approach. We base the attributes on the metaclasses in the reference metamodels as well as on the different representations of those metaclasses in our analyses.

- **A1: View on the System:** Analyses use system elements of the *architecture* or *implementation* to detect vulnerabilities. This attribute allows to differentiate between architectural and source code analyses. In our case study, we select multiple analyses of the same view 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 2.2). One particular case is the use of *no* values of security characteristics of data in source code analyses, e.g., as in CodeQL [65]. Source code analyses using *no* values detect information flows only between specified sources and sinks. The importance of this attribute is that our ICs influence the security characteristics of data. Furthermore, any source code analysis with *predefined* values will also constrain the values usable in the architectural specification.
- **A3: Configuration Representation:** Analyses may use either *explicit* configuration elements or an *implict* configuration (see Section 6.1.3). 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* in the *Implementation Value Integration* step. Consequently, wherever an *implicit* configuration is used by an analysis in a coupling scenario, it becomes necessary to realize (e.g., by coupling tool) definitions for exactly which analysis in the coupling uses or provide exactly which in- or output model.
- **A4: Policy Specification:** The security policy is either *configurable* by the analysis user or it is *integrated* in the analysis technique (see Section 2.2). For the values of the security characteristics to be mapped correctly and for

TABLE 1
Selected Analyses for Case Study. 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.).

Analysis	View	Values	Config. Rep.	Pol. Spec.
Access Analysis [21]	Arch.	Spec.	Implicit	Integr.
DFA [6]	Arch.	Spec.	Implicit	Config.
DFA-Ext	Arch.	Spec.	Implicit	Config.
JOANA [23]	Code	Spec.	Explicit	Config.
CodeQL [65]	Code	Not	Explicit	Config.
CodeQL-Ext	Code	Spec.	Explicit	Config.

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:** The input model of an architectural analysis is defined to comprise specifications of security characteristics of data which are influenced by the information flows in the system; whereas, the source code analysis uses security characteristics of data to detect non-compliant information flows in the implementation.
- **A6: Working Tooling:** We require every analysis to provide *Working Tooling* as we must be able to execute the analyses for answering **Q2**. 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 found only a few architectural and source-code analyses with *Working Tooling*.

Table 1 shows this classification of analyses by attribute. We omit the two attributes *Information Flow Domain* and *Working Tooling* because they are mandatory attributes in every analysis of our case study.

We use the Access Analysis [21] as architectural analysis (**A1**). Access Analysis uses *specifiable* ranges of security characteristics of data (**A2**), first, to specify expectations about internal information flows, and second, to use those flows to calculate whether adversaries can gain unallowed access to input data. Access Analysis uses an *implicit configuration* because only the input models are used in the analysis (**A3**). The security policy of the Access Analysis is *integrated* (**A4**) because of how unallowed access is evaluated: Prolog clauses use, amongst others, the *DataSets* assigned to parameters of services to determine whether an adversary can gain unallowed access to data allocated to those parameters.

We use an extended version of DFA [6] 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 such as parameters. This extension of DFA is made possible by the fact that the condition for invalid flows is configurable in the analysis tooling of the DFA [63]. We 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 [6] (detailed in Supplementary Material [68]). 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**), it uses no specific configuration element besides the input models (i.e., *implicit* configuration (**A3**), and its security policy is configurable by defining unallowed data flows between values (**A4**).

Although numerous source code information flow analyses are available in literature (e.g., [10], [12], [23], [24], [25], [65]), only a few of them have readily available, automated, and working tools. 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 [65] or FlowDroid [12] 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* [23] and CodeQL [65] for our case study because they provide working tooling.

JOANA is used in related work [29]. *JOANA* 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**). *JOANA* is limited to the programming language Java.

CodeQL [65] detects flows between sources and sinks, provides a configurable output, and also allows the analysis user to define a *FlowState* for the information flow analysis. This *FlowState* 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*. Consequently, the CodeQL information flow analysis does not fit the purpose of investigating information flows based on security characteristics of data (**A5**). However, CodeQL's input, 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.

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*). 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*). To evaluate the input and output of *CodeQL-Ext*, we created EMF [81] metamodels. Accordingly, the source code analysis (**A1**) *CodeQL-Ext* utilizes *specifiable* value ranges for security characteristics of data (**A2**). Like

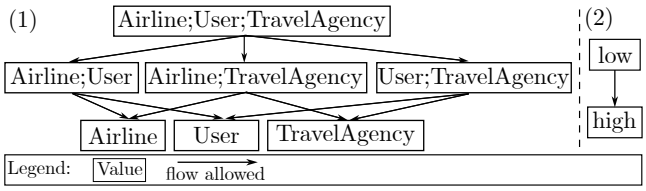

Fig. 6. Examples of applied classes of case study information flow lattices for (1) TravelPlanner (reduced set of values) and (2) JPMail.

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 please see Supplementary Material [68]

9.2.1.2 **Selection of Systems**: To select the systems in our case study, we use the semantics distinctive to each system in the values of the security characteristics of data. In two of our case study systems, the security characteristic of data represents a *role* in a role-based access scenario with disjunctive lattice; but in the other two systems, 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 Fig. 6. The value of a role may be a combination of basic values, but the value of a confidentiality level may not. A disjunctive 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 [72])

For every system, we require an architectural model, a security specification, and an implementation. *JOANA* and CodeQL are whole-program analyses that calculate the information flows for monolithic programs with call-return semantics. This is why Katkalov et al. [29] generate source code in a single code base for the information flow analysis with *JOANA*, although they design message-oriented applications. Thus, 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, see Supplementary Material [68]; and for the actual architectural models and implementations see the replication package [72].

The system we selected are *TravelPlanner* [6], [82], *JPMail* [6], [19], *EclipseSecureStorage* [28], [44], [83], and CoCoME [84]. 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. Fig. 6 shows 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.

TravelPlanner enables users to book flights. To describe the roles an adversary must fulfill to be allowed access to specific data, the TravelPlanner specification [82] assigns role-based values like *TravelAgency*, *User*, and *Airline* to data (security characteristic of data). To establish the ground truth (described in Section 9.2.3), we add the role *Support* (omitted from Fig. 6 for simplicity). For all roles and their combination, TravelPlanner uses a *disjunctive lattice*. TravelPlanner is used in related work to evaluate IFlow [29], DFA [6], and Access Analysis [21]. This makes TravelPlanner relevant to our evaluation. The architectural models of TravelPlanner already exist for DFA [6] and Access Analysis [21], so we can use these in the evaluation. Further, the IFlow approach [29] is also relevant to our coupling because IFlow enables analysis of non-compliance between the designed system and the implementation.

CoCoME [84] is a community case study for component-based systems which describes a TradingSystem in supermarkets that manage sales using cash desks. CoCoME was extended by Greiner and Herda [85] — 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 has a *disjunctive lattice* for roles. Related work use CoCoME or CoCoME with Security to evaluate information flow analyses [25], [37] or compliance analysis [28]. Therefore, CoCoME and CoCoME with Security are relevant for our evaluation. Greiner and Herda [85] 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 [85]) using information from the original CoCoME description [84] 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 9.2.3). Therefore, to establish an initial system without any security vulnerabilities, we modify the architectural models and the implementation (please see Supplementary Material [68] for all details). Henceforth, we refer to CoCoME with Security simply as CoCoME.

JPMail is a message-oriented email exchange system. JPMail is used in related work to evaluate SecDFD [19], [28] and DFA [6]. 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. [19] specify the security characteristic *Attack Zone* and data encryption.

EclipseSecureStorage [83] enables the storage of passwords by Eclipse Plugins and also the recovery of lost master passwords via a challenge response procedure. EclipseSecureStorage is used in the evaluation of SecDFD [28] to assess compliance between designed and implemented information flows. This makes EclipseSecureStorage also relevant to our evaluation. Similar to JPMail, SecDFD 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, 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 Supplementary Material [68]).

9.2.2 Specialization of Coverage Metric

Here, we adapt our *Coverage* metric to evaluate the coverage of the reference metamodels (**Q1.1**) and the coverage of the integration conditions (**Q1.2** and **Q1.3**) (see Section 9.1). For simplicity, we omit the parameters of $hold$ and $Coverage$ wherever appropriate and provide a formalized version of this specialization in Supplementary Material [68].

9.2.2.1 **Coverage of Reference Metamodels**: To evaluate the coverage of the reference metamodels to answer **Q1.1**, we construct $Coverage_{Ref}(A_{Ref}, COND_{Ref})$. Here, we calculate the coverage by utilizing the metaclasses and references of the input or output metamodels of the analysis under study (A_{Ref}). For each metaclass of the reference metamodels, we define a condition stating that this metaclass has a conforming metaclass in the input or output metamodel of an analysis ($COND_{Ref}$). The reference metamodel for output metamodels is described only for source code analyses. Consequently, we only define conditions for the reference metamodel for output metamodels and consider the output metamodels of the analysis in the evaluation artifacts when inspecting source code analyses. To calculate the coverage, we define $hold_{Ref}$ to return 1 if a metaclass exists in the metamodels of the analyses that *conforms* to the given metaclass of the reference metamodels in the condition as stipulated by our Definition 4. Otherwise, $hold_{Ref}$ returns 0. Thus, we define the reference metamodels to be fully covered by an analysis if $Coverage_{Ref}$ evaluates to 1, that is, if for every metaclass of our reference metamodels, a conforming metaclass is found 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 so as 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 analysis to make the analysis usable with our coupling approach, thus contradicting a black-box coupling.

9.2.2.2 **Coverage of Integration Conditions**: To answer **Q1.2** and **Q1.3**, we construct the metric $Coverage_{IC}$ which evaluates the coverage of the ICs on the metamodel and model level. We define a condition for each IC stating that it has to hold on the metamodel and model level ($COND_{IC}$). The ICs describe relations between the input and output (meta)models of the architectural and source code analyses. The ICs are also formulated for the *Resolved Implementation Values Model*. In our evaluation, each transformation generates a correspondence model capturing the correspondences between the elements in the transformations of our coupling process. Therefore, we consider the input and output metamodels of the analyses, the metamodel of the *Resolved Implementation Values Model*, the correspondence metamodels, and the respective models as evaluation artifacts (A_{IC}).

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. Only investigating whether metaclasses conform to the ICs (e.g., by capturing correspondences) 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.

Thus, $hold_{IC}$ 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) if 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 *Level* and the instance *high* of the metaclass *Security Characteristic of Data* (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 *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* of such an 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 defintion 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 $Coverage_{Ref}$ 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.

9.2.3 Definition of Ground Truth and Result Classification for Accuracy Metrics

For the calculation of *Precision* and *Recall* to evaluate accuracy (G2), the results of an analysis must be classified into true positives, false positives, and false negatives. For this classification, we require (1) knowledge about the expected results of an analysis (ground truth), and (2) a comprehensive definition of true positives, false positives, and false negatives with regard to the expected results. To begin, we obtain knowledge of expected results by systematically injecting information

flows into the implementation of the case study systems. In this way, we obtain values of the security characteristic of data that are non-compliant with the architectural model in the coupling scenario. The values thus obtained affect the result provided by the architectural analysis. We create the architectural models so that an isolated architectural analysis does not report any vulnerabilities, finally making the analysis perform only reports based on the coupling. Our procedure here relates closely to the evaluation of accuracy in related work [6], [19], [28], [29], [44].

9.2.3.1 **Injected Information Flows**: To guide our injection of information flows, we use three equivalence classes (**EC1**, **EC2**, and **EC3**) intended to influence the coupling and the result of the architectural analysis. We inject information flows so that each equivalence class is covered at least once in at least one of the cases. Through the injected information flows, we demonstrate the capabilities and shortcomings of our coupling approach.

EC1	**Description:**	*Values of security characteristics of data flowing to an implementation element*
	Categories:	*same, different*

EC1: The accuracy of the coupling highly depends on the calculation of the values of security characteristics resulting from the implementation in the *Implementation Value Integration* 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 **EC1** with the categories *same* and *different*. 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; here, 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.

EC2	**Description:**	*Effect on architectural result*
	Categories:	*effect, no effect*

EC2: As our running example illustrates, a change of the values specified in the *Annotated Architecture* by the coupling due to non-compliance does not necessarily lead to new vulnerabilities. Our equivalence class **EC2** captures the effect which an information flow will have on the architectural analysis result. **EC2** contains the categories *effect* and *no effect*. For information flows of the category *effect*, the architectural analysis reports additional vulnerabilities due to the coupling. For information flows of the category *no effect*, new values of the security characteristic of data are calculated and represented in the *Annotated Architecture* such that the values do not affect the architectural analysis result. **EC2** allows investigating the effect on the architectural analysis result of those non-compliances detected between implementation and architectural model.

EC3	**Description:**	*Visibility of sources of an information flow*
	Categories:	*visible, hidden*

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.

We inject seven information flows into TravelPlanner, six into JPMail, one into CoCoME, and one into EclipseSecureStorage based on **EC1** to **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 2 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., [6], [19], [29], [35]). *At least one* injected information flow of this type is found in each of our case study systems. In Table 2,

TABLE 2
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).

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

```
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 2. 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.

we list the number of expected architectural vulnerabilities reported after applying the coupling. Based on 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.

We now describe examples of the injected flows. All injected flows are detailed in Supplementary Material [68].

In TravelPlanner, we inject an information flow already used for evaluating the IFlow approach [29] and the DFA [6]. This information flow originates 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* by omitting a declassification. 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 service *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 service *addToFlight*, is not accessible by any actor. Access Analysis calculates vulnerabilities by considering required and provided interfaces of components, but DFA-Ext considers only provided interfaces. Thus, 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.

In EclipseSecureStorage, we inject a single information flow from the *visible* (**EC3**) parameter *challengeResponse* of the service *setupRecovery* provided by the *PasswordManagement* class, classified with the value of security characteristic of data *high*. This information flow has already been investigated by Tuma et al. [28]. As illustrated in Listing 2, we remove the encryption of the password derived from the answers in *challengeResponse* to provide, instead, the password directly to the service *internalPut* of the *SecurePreferences* class by the parameter *value* classified as *low*. This flow leaks the *same* security characteristic of data (**EC1**), from the *high* classified *challengeResponse* of the service *setupRecovery* to the *low* classified *value* of the service *internalPut*. This injected flow has an *effect* (**EC2**) because we model *SecurePreferences* and its services to be accessible by an *Attacker*.

9.2.3.2 **Definition of True Positives, False Positives, and False Negatives**: Here, we define our classification of vulnerabilities reported by the architectural analysis after the coupling into true positives (t_p), false positives (f_p) and false negatives (f_n). The categories *effect* and *no effect* of equivalence class **EC2** describe that the injected information flows must (*effect*) or must not (*no effect*) result in reports of the architectural analysis when values of a security characteristic are changed by the coupling. These two categories result in different definitions of t_p, f_p, or f_n.

To begin, we define the t_p, f_p and f_n for the category *effect* (i.e., vulnerability is expected) as follows. A t_p is a vulnerability that is *reported* by the architectural analysis and is *expected* due to the injected information flows. A f_p is a vulnerability that is *reported* by the architectural analysis and is *not expected* due to the injected information flows. This definition of a false positive is conservative. The coupling may have in the architectural models modified values of a security characteristic of data which validly result in a vulnerability but 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. A f_n is a vulnerability *not reported* by the architectural analysis but is *expected* to be reported based on the injected information flows.

We define the t_p, f_p and f_n for the category *no effect* (i.e., vulnerability is *not* expected) as follows. A t_p is when *no* vulnerability is *reported* by the architectural analysis as *expected* due to the injected information flows. Our running example describes a situation where an information flow results in the value of the confidentiality level *medium* for *userData*, but that value is specified in the architectural model as *low*. No vulnerability is reported due to the coupling, because the security policy allows *medium* data to be allocated on *Non-EU* resources. Consequently, we would count this outcome as t_p. A f_p is when *no* vulnerability is *reported* by our

TABLE 3
Results for the *Coverage* of the reference metamodels by the input and output metamodels of the case study analyses. 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.

	Access Analysis	DFA	DFA-Ext	JOANA	CodeQL	CodeQL-Ext
Input	1	1	1	1	0.2	1
Output	-	-	-	0.8	0.4	1

architectural analysis although we expect a vulnerability due to the injected information flows. A f_n is when a vulnerability is *reported* by the architectural analysis, but this vulnerability is *not expected* based on the injected information flows.

9.3 Evaluation Results and Discussion

We present and discuss the evaluation results of our metrics *Coverage* of the reference metamodels and ICs in Section 9.3.1. In Section 9.3.2, we present and discuss the evaluation results of our metric *Accuracy*.

9.3.1 Result Presentation and Interpretation: Coverage

Here we report and interpret our results to answer the questions **Q1.1**, **Q1.2**, and **Q1.3** for our evaluation of **G1**. We report the results separately for the coverage of reference metamodels (**Q1.1**) and the coverage of ICs (**Q1.2** and **Q1.3**); however, we interpret all results together, to highlight the interplay of the reference metamodels and ICs.

9.3.1.1 **Results for Coverage of the Reference Metamodels**: Table 3 presents our results for the metric $Coverage_{Ref}$, as we calculated it for six input and three output metamodels. The metric is calculated for the four selected analyses detailed in Section 9.2.1 but also for CodeQL and DFA to investigate the coverage of the reference metamodels by metamodels that are not usable in our coupling.

To exemplify one case where the reference metamodels are fully covered and also one where they are not, note how the Access Analysis input model was fully mapped ($Coverage_{Ref} = 1$), but the CodeQL input model was not ($Coverage_{Ref} = 0.2$). For the detailed assertion to the metaclasses of the reference metamodels to the metamodels of the analyses, please see Supplementary Material [68].

Each reference metamodel 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 ($Coverage_{Ref} = \frac{1+1+1+1+1}{5} = 1$).

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 Section 6.1.3). The Prolog rules of Access Analysis define unallowed constellations between elements annotated with security characteristics to determine when a vulnerability is reported (*integrated Security Policy*). For example, the Prolog rules define how 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.

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*). Since CodeQL still uses *System Elements* in its input metamodel, $Coverage_{Ref} = \frac{1}{5} = 0.2$. The metric *Coverage* of the reference metamodel for the output metamodels is $Coverage_{Ref} = \frac{1+1}{5} = 0.4$ 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: This metamodel does not fully cover the reference metamodel for output metamodels because the metaclass for the *Result Entries* does not reference the metaclass EntryPoint (*Configuration*), but vice versa. 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. 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.

9.3.1.2 **Results for Coverage of Integration Conditions**: We omit a table which represents the results for $Coverage_{IC}$ because $hold_{IC}$ evaluated to 1 for all ICs in all sixteen cases on both the model level and the metamodel level. This means that $Coverage_{IC} = 1$ (full coverage) of all ICs in all cases. For illustration, we describe the calculation of $hold_{IC}$ for **IC1** and **IC2** (*correspondence conditions*) and for **IC3** (*constraint condition*) on the case comprising Access Analysis, *JOANA*, and JPMail. However, detailing the results for every IC and every case would exceed page limitations, as this comprises 360 applications of $hold_{IC}$ (16 cases 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 Supplementary Material [68] and the replication package [72].

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*. Thus, $hold_{IC}(A_{IC}, IC1)$ yields 1 on the metamodel and model level. Also, all annotated instances of *Parameter* in the *Annotated Source Code* are captured in a correspondence to *Parameters* in the *Annotated Architecture*, e.g., 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 cases, **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.

9.3.1.3 **Result Interpretation**: The results of our metrics *Coverage* show that all elements of the reference metamodels are fully covered (i.e., $Coverage_{Ref} = 1$) by the input and output metamodels of Access Analysis, DFA, DFA-Ext, and CodeQL-Ext. In the calculation for CodeQL, the element *Security Characteristic* of the reference metamodels caused the metric *Coverage* of the reference metamodels to decrease: $Coverage_{Ref} = 0.2$ for the input metamodel and $Coverage_{Ref} = 0.4$ for the output metamodel). By way of contrast, the metric *Coverage* of the reference metamodels for the DFA input metamodel is 1 despite the need for its extension to be applicable in the coupling. 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* ($Coverage_{Ref} = 0.8$), because the reference direction is inverted between the metamodel element *EntryPoint* (*Configuration*) and the *ResultEntry*. Semantically, the reference 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., $Coverage_{Ref} = 1$). In the replication package [72], 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 $Coverage_{Ref}$ shows that the reference metamodels are indeed fully covered by the input and output metamodels of the analyses. Therefore, we answer **Q1.1** with *yes*. The reference metamodels are covered by the input and output metamodels of architectural and source code analyses to detect security vulnerabilities using information flow-related security characteristic of data. Furthermore, since $Coverage_{IC} = 1$ on the metamodel level for all cases, we answer **Q1.2** too with *yes*. Finally, since $Coverage_{IC} = 1$ for every successful coupling in our case study, we also answer **Q1.3** with *yes*. We can resolve all model elements according to our conditions; hence, ICs are covered on the model level. Again, since all ICs hold in every successful coupling on the metamodel and model level, we conclude that the ICs are necessary conditions. Two insights here are worth noting.

TABLE 4
Accuracy results by true positives t_p, false positives f_p, false negatives f_n, precision p, and recall r for all cases (in the first column). For system abbreviations, see Table 2.

Case	t_p	f_p	f_n	p	r
(AccessAnalysis, CodeQL-Ext, CCM)	$\frac{3}{3}$	0	0	1	1
(AccessAnalysis, CodeQL-Ext, ESS)	$\frac{1}{1}$	0	0	1	1
(AccessAnalysis, CodeQL-Ext, TP)	$\frac{7}{7}$	0	0	1	1
(AccessAnalysis, CodeQL-Ext, JP)	$\frac{8}{8}$	0	0	1	1
(AccessAnalysis, JOANA, CCM)	$\frac{3}{3}$	0	0	1	1
(AccessAnalysis, JOANA, ESS)	$\frac{1}{1}$	2	0	$\frac{1}{3}$	1
(AccessAnalysis, JOANA, TP)	$\frac{7}{7}$	1	0	$\frac{7}{8}$	1
(Access Analysis, JOANA, JP)	$\frac{8}{8}$	0	0	1	1
(DFA-Ext, CodeQL-Ext, CCM)	$\frac{2}{2}$	0	0	1	1
(DFA-Ext, CodeQL-Ext, ESS)	$\frac{1}{1}$	0	0	1	1
(DFA-Ext, CodeQL-Ext, TP)	$\frac{4}{4}$	0	0	1	1
(DFA-Ext, CodeQL-Ext, JP)	$\frac{4}{4}$	0	0	1	1
(DFA-Ext, JOANA, CCM)	$\frac{2}{2}$	0	0	1	1
(DFA-Ext, JOANA, ESS)	$\frac{1}{1}$	2	0	$\frac{1}{3}$	1
(DFA-Ext, JOANA, TP)	$\frac{4}{4}$	0	0	1	1
(DFA-Ext, JOANA, JP)	$\frac{4}{4}$	0	0	1	1

(1) We extend CodeQL and DFA because they could not be used in the coupling. Although CodeQL showed $Coverage_{Ref} < 1$ because it does not represent security characteristics of data, DFA actually showed $Coverage_{Ref} = 1$ and still we need to extend it. Since DFA does not annotate parameters with security characteristics, it cannot be coupled with *JOANA* and CodeQL-Ext. Thus, conformance to the reference metamodels alone does not guarantee applicability in the coupling because *relations* between analyses may be lacking. Therefore, the ICs are necessary.

(2) Couplings with *JOANA* were successful despite the fact that the output metamodel has a $Coverage_{Ref} < 1$. To enable the coupling, we provided a transformation to a metamodel that is semantically equivalent to the output of *JOANA* and conforms to our reference metamodels. We conclude, based on this case, a coupling can be established despite $Coverage_{Ref} < 1$ only if such semantically equivalent transformations and metamodels can be provided by the *Coupling Expert*.

9.3.2 Result Presentation and Interpretation: Accuracy

Here we report and interpret our results to answer **Q2** for our evaluation of **G2**.

9.3.2.1 **Results for Accuracy**: Table 4 presents our results for the metric *Accuracy* for all sixteen cases. Presenting and discussing all results in detail would exceed space limitations. Thus, we provide examples of (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 Supplementary Material [68] and

in the replication package [72], we provide all changes of the security characteristics of data and the resulting architectural vulnerabilities reported by the architectural analysis.

In Table 4, the t_p column shows that in all sixteen cases, the architectural analyses reported the vulnerabilities as expected due to the injected information flows (detailed in Section 9.2.3). 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 from *Airline* to *User* that are annotated to the parameter *ccd_decl* in the service *bookFlight* of the component *Airline*. 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 Section 9.2.3), 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) any vulnerabilities other than the ones expected due to the injected information flows. Thus, it is recall $r = 1$ for all cases.

Five false positives have been reported after the coupling in three of the sixteen cases (see f_p column), resulting in a precision $p < 1$; all other cases have $p = 1$. 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. The DFA-Ext did not report the vulnerability because it does not consider the respective interfaces. 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 cases). 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 f_p vulnerability was found for the parameter *pathName* in the provided *node* service of the *SecurePreferences* component.

9.3.2.2 **Result Interpretation**: In total, three of our cases — all involving JOANA — reported f_p yielding $p < 1$. All other thirteen cases yielded $p = 1$. We can exclude scenarios where the false positives arise from unanticipated *data flows*, because CodeQL-Ext did not report the flows leading to f_p. Our explanation for these f_p is either that they are caused by implicit information flows we did not comprehend or that they are 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 cases, information flows for all parameters of the method are reported instead of only one. 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 those results 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 Section 9.1). From this we conclude that the output of the source code analysis directly influences the accuracy of the architectural analysis in the coupling. The f_p 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.

Yet, in all other cases of our evaluation, the coupled analysis found all expected vulnerabilities or did not report those of the class *no effect*. Nonetheless, it was possible to identify vulnerabilities *fields* (category *hidden* (**EC3**) described in Section 9.2.3) 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 sixteen cases, we found 5 f_p compared to a total of 60 t_p. Based on this evidence, we answer **Q2** with *yes*; that is, the coupled analyses are indeed accurate regarding vulnerabilities arising from non-compliant information flows.

9.4 Threats to Validity

The validity of our evaluation is subject to different threats. We organize our discussion here by the threat categories *Internal Validity*, *External Validity*, *Construct Validity*, and *Reliability* of Runeson et al. [86].

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 are the coverage of the reference metamodels, the coverage of the ICs, and the accuracy of the architectural analysis in the coupling.

We evaluated the coverage of the reference metamodels using our metric $Coverage_{Ref}$. The validity of this metric is influenced by two variables: our selection of the analyses and the mappings created between the reference metamodels. To counter any threat posed by our selection of the analyses in the case study, we generalize the analyses by constructing a set of attributes such that each attribute from the total set is used at least once in every analysis. 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*) by studying the available documentation of Access Analysis [21], *JOANA* [23], and DFA [6]. Wherever the documentation was unavailable or proved insufficient to task, we contacted the authors or maintainers of the analyses. We can exclude this threat for CodeQL-Ext because we are the creators of the metamodel.

We evaluated the coverage of the ICs using our metric $Coverage_{IC}$. The metric is influenced by two variables: the transformations created between the (meta)models of the analyses and the realization of $hold_{IC}$. The transformations created yield, amongst other things, the correspondence models we investigated and as well, 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. Therefore, to counter this threat, we precisely define, for each of our sixteen cases, 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 Supplementary Material [68]). By thus applying the same evaluation logic of $hold_{IC}$ for each IC for all sixteen cases, our automation prevents overfitting the evaluation to one case while thereby increasing the comparability of results.

Our evaluation of accuracy is mainly influenced by experimenter bias in the creation of a ground truth ([28]). Our ground truth comprises the architectural models, the security specifications, the implementation, and all injected information flows which affect the values of security characteristic of data in our architecture. In fact, the implementation 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 inadvertently affected by our implementation if we reuse an existing implementation or reimplement the system by using existing information flows. This is also true when performing errors in creating the architectural model and the specification. 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. [6]; for TravelPlanner, the model of Kramer et al. [21] and the behavior description at [82]; for CoCoME, the PCM model of Greiner and Herda [85]; and for Eclipse Secure Storage, the implementation of Tuma et al. [28]. However, our evaluation is also threatened by the fact that we are the ones who inject the information flows into the implementation to affect the values of security characteristic of data in the architectural input model. Therefore, to counter this threat, first we systematically inject at least one illegal information flow in every system by omitting a declassification. This is common procedure in literature [6], [19], [29], [44], and it has been applied for our case systems TravelPlanner ([29]) and JPMail ([6], [19]). Location of the declassification has also been described for our case systems EclipseSecureStorage ([28], [44]) and CoCoME ([85]). 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 Section 9.2.3). One final threat here is the influence potentially exerted by the quality of a transformation in the coupling, especially the quality of the retrieval of values of the security characteristic and its integration in the *Implementation Value Integration* step. Therefore, to counter this threat, we use case systems with different semantics of their security characteristic to account for different implementations of transformations. For example, JPMail has confidentiality levels with the values *high* and *low* and a *strictly ordered lattice*, and TravelPlanner uses roles with the values *User*, *TravelAgency*, and *Airline* with a *disjunctive lattice*.

9.4.2 External Validity

External Validity concerns the generalizability of our findings beyond just the cases 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 [86]. In this way, a case study design allows us to gain valuable indicators for cases with similar properties [86].

External validity also concerns the analyses we selected, the security characteristic we investigated, the architectural models we created, the implementation we conducted, and the information flow-related security characteristics we specified in the cases. Since there may be other cases that may lead to other results, the thing particularly at risk here is the external validity of our evaluation of coverage and accuracy. Therefore, to counter this threat, first we classified our analyses based on attributes with regard to our reference metamodels. Then we selected (as far as possible) a representative analysis for each combination. 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 [6], [19], [29] 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.

One other threat to external validity is the limited generalizability of the information flows we injected. To counter this threat, 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 [29], for CoCoME [85], for JPMail [6], [19] for Eclipse Secure Storage [83]).

9.4.3 Construct Validity

Construct Validity ensures that metrics are capable of answering the evaluation questions. The metrics we applied are $Coverage_{Ref}$ to evaluate the coverage of the reference metamodels, $Coverage_{IC}$ to determine the coverage of our ICs, and $precision$ and $recall$ to evaluate the accuracy of the architectural analysis in our coupling. We discuss the appropriateness of each metric to each question in Section 9.1. 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 Section 6). 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 [67] for further details.

Construct validity also concerns the potential overfitting of the IC. To counter this threat, 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 and only then we selected the other cases.

Our metrics $precision$ and $recall$ may pose threats by being inappropriate for measuring increases in accuracy. We are able to exclude this threat because $precision$ and $recall$ are both widely used in related work (e.g. [6], [12], [28], [36], [44] to measure the accuracy of the analysis under study.

One other threat to construct validity is the reporting of vulnerabilities which *do not* arise from the coupling. To counter, we first ensure that the architectural analysis does not report any vulnerabilities (cf. [44]) before performing the coupling. In this way, we ensure that we consider only those vulnerabilities arising from the coupling.

9.4.4 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 Supplementary Material [68] and in the replication package [72]. 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. Furthermore, we increase reliability by automating the calculation of $hold_{IC}$, which determines the coverage of the ICs (detailed in Supplementary Material [68] and in replication package [72]). 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 supplementary material [68] and replication package [72]: 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 ([6], [19], [29]). Our metrics *Coverage* and *Accuracy* (detailed in Section 9.2) give reasonable evidence, which, in turn, reduces the necessity for interpretation. Thus, due to the study design in Section 9.2, there is hardly an interpretation that may lead a researcher to another conclusion.

10 Related Work

Related work to our coupling approach can be classified in the following categories: (a) detecting non-compliance between design models and implementation for security by analysis, (b) lifting analysis results on another abstraction level, (c) coupling of analyses. These categories are relevant because the first two categories present related work similar to the steps of our coupling process, i.e., the *Analyses Input Model Consistency* and *Non-Compliance Analysis* (a) and the *Integration* transformation (b). We provide a coupling approach for analyses, so we discuss related work regarding analysis coupling (c). We discuss the related work in these categories accordingly. While our coupling specifically targets architectural models, we also consider other design-level approaches, e.g., data flow models or Unified Modeling Language (UML) models such as sequence diagrams.

10.1 Detecting Non-Compliance between Design Models and Implementation for Security by Analysis

One topic in related work is the analysis of conformance between design models and implementation, which is realized with the *Analyses Input Model Consistency* and *Non-Compliance Analysis* steps in our coupling approach. However, approaches in related work are tailored to specific models and tools, while we abstract from them by applying reference metamodels and ICs. In addition, these approaches do not consider the *Implementation Value Integration* step and how the information gained by the source code analyses relates to the architectural model, leaving the interpretation of detected non-conformances and the relation to the architectural analysis up to the software engineers.

CARDS [18], IFlow [29], Töberg et al. [46], Muntean et al. [87] and Yurchenko et al. [88] analyze specification-based non-compliance by deriving specifications for information flow analyses and also by mapping design models to source code. SecDFD [28] analyzes design-related non-compliance by creating mappings between data flow diagrams and source code to investigate whether the design and implementation converge or diverge.

10.2 Lifting Analysis Results on Other Abstraction Level

Different approaches are concerned with lifting the results of an analysis to another level of abstraction. Several related works first transform a user-facing design model into an analysis formalism, then they perform an analysis with this formalism, and finally they lift the results back into the abstraction level of the user-facing model. This lifting is performed either by presenting the results with model elements of the user-facing model ([87], [89], [90]) or by refactoring the original user-facing model ([91], [92]). In contrast, our coupling addresses artifacts of different development phases (i.e., architecture and implementation) rather than just artifacts of a single phase. This allows the detection of vulnerabilities that cannot be detected by only considering artifacts of one abstraction level. Furthermore, we aim to provide insights into the impact of non-compliance between the architecture and code. iObserve [53] and CIPM [93] 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. GraViTY [44] removes the *secure links* annotation of UMLSec [17] when non-compliance is detected with the source code analysis FlowDroid [12]. We provide an approach independent of specific analyses and security characteristics to better guide the coupling of other pairs of analyses without the need for coupling experts to understand a description of a specific coupling.

10.3 Coupling of Analyses

The coupling of analyses is another related topic. In contrast to related work, we provide a black-box coupling approach and concrete process to couple architectural analyses with source code analyses for information flow. OPAL [59] is a grey-box approach for coupling static program analyses, which requires the integration of framework functionality into the analysis tooling to enable communication. Our black-box coupling avoids such modifications, which are beneficial regarding maintainability and portability (see Section 4.2). Dwyer et al. [94] and Cruanes et al. [95] define black-box coupling approaches, which focus on making the results of different tools available to each other without detailing how analyses can interact with each other. In contrast, we define the concrete process in Section 5, which allows coupling experts a more guided application of our approach.

11 Limitations and Future Work

Our approach is subject to several limitations from which we derive future work.

Our approach is limited by its focus on information flow security. Information flow security is an important topic in research and industry, as motivated in Section 1, which makes the focus on this topic acceptable. However, future work can comprise research on extending the coupling approach to analyze security characteristics other than those related to information flow. Further, our ICs and reference metamodels form a coherent foundation for creating the transformations in the coupling. However, we provide no constructive guidance for creating the transformations, which is also important to support coupling experts in creating the coupling. Therefore, future work can involve researching whether we can provide constructive guidance for creating the transformations in the coupling process. Our reference metamodels and ICs support the *coupling expert* in deciding whether a coupling can be established based on the (meta)models of the analyses. However, the selection of a source code analysis with capabilities fitting to the envisioned coupling scenario remains yet a fully manual task. Thus, we will research how to relate architectural analyses with source code analysis based on the analysis capabilities.

Additionally, as described in Section 9.1, we are aware that the accuracy of the source code analysis can negatively impact the result of the architectural analysis in the coupling. A research direction could be to investigate whether this issue can be addressed in the coupling approach.

We also showed that changes in values of security characteristics in the architectural model may or may not result in the detection of new vulnerabilities. However, we did not investigate how software engineers can use this insight in the later development process. Future work in this direction will investigate how software engineers can use relations between new vulnerabilities in the architectural analysis and non-compliance in the implementation.

The role *coupling expert* described in Section 4.3 aims to separate the responsibilities further to allow specialization and reduce errors, which is common in software engineering processes. Still, we see that the tasks are still error-prone when performed manually. Future work can comprise research on how automation can cover the tasks of the *coupling expert*, such as extracting correspondences conforming to *correspondence conditions* with Large Language Models.

Currently, the design of our coupling approach only considers non-compliant values resulting from the implementation due to *existing* information flows. Still, non-existent information flows can also result in non-compliance, e.g., when confidential data does not flow to a parameter despite being specified otherwise. Such non-compliance can result in the report of an architectural vulnerability not existing in the final system. In future work, we will research how to use coupling to consider this type of non-compliance.

We focus only on static analysis to detect non-compliance, which avoids deployment of an insecure system. This does not allow for the comprehensive investigation of all security aspects or dynamic scenarios. Future work can comprise research on how to couple design analyses with other types of implementation-level analyses, such as dynamic analysis.

12 Conclusion

Architectural analyses may not detect security vulnerabilities because of invalid assumptions about the compliance of the values of security characteristics realized by the implementation and those specified in the input model of the architectural analysis. We address this problem by providing a coupling approach for coupling an architectural analysis and a static information flow source code analysis with the aim to enrich the architectural analysis input model with the values of security characteristics resulting from the implementation, uncovered with a source code analysis. With this, the architectural analysis can be performed based on values of security characteristics resulting from the implementation rather than based on values specified by a software engineer early in the design phase. The approach performs a black-box coupling by utilizing the input and output models of the analyses and transformations between them while avoiding modification of the analyses algorithms or their tools. Our coupling approach comprises two contributions: Contribution **C1** provides the coupling process comprising the input and output models involved in the analyses, transformations between these models, and the responsibilities of roles in the creation of analyses. The coupling is only possible if the information about the values of the security characteristic resulting from the implementation can be integrated into the architectural model. Contribution **C2** provides Integration Conditions (ICs) and reference metamodels for the input and output metamodels/models of the analyses to achieve a coupling independent of specific analyses and security characteristics. The ICs form a set of necessary conditions capturing relations between the input and output metamodels/models of the analyses in the coupling that have to hold

so that an integration of the values of a selected security characteristic resulting from the implementation into the architectural model is possible. These ICs are described based on reference metamodels, which abstract the elements of the input and output metamodels of specific analyses.

We perform a case study-based evaluation examining two goals to evaluate our contributions: We evaluate **C2** by inspecting the coverage of the reference metamodels by the input and output metamodels of the analyses and the coverage of the ICs by successful couplings (**G1**). We also evaluate **C1** by inspecting the accuracy of the coupled analysis regarding architectural vulnerabilities arising from non-compliant information flows in the implementation (**G2**). In the evaluation of **G1**, we found that the reference metamodels are covered by the input and output metamodels of architectural analyses investigating information-flow related security characteristic of data and source code analyses investigating information flow. Further, we found full coverage of all ICs on the metamodel and model levels in the examined coupling scenarios, i.e., the conditions hold in successful couplings. We also discuss insights obtained regarding the relation between the reference metamodels and ICs. In the evaluation of **G2**, the coupled analyses reported 60 true positive and five false positive vulnerabilities. These results show that our coupled analyses applied in the case study has good accuracy in detecting vulnerabilities arising from a non-compliant implementation.

We conclude that we successfully addressed the problem of architectural vulnerabilities not detected with an architectural analysis because of an implementation that does not comply with the architectural specification. We also conclude that our ICs and reference metamodels are covered in the coupling of different architectural analyses and source code analyses, providing a generalized approach for their coupling.

ACKNOWLEDGMENTS

This work was supported by funding from the project FeCoMASS by the German Research Foundation (DFG) under project number 499241390 and the topic Engineering Secure Systems of the Helmholtz Association (HGF) and by KASTEL Security Research Labs (46.23.01 Methods for Engineering Secure Systems). Thanks to our textician.

REFERENCES

[1] "Metrics | CVE." [Online]. Available: https://www.cve.org/about/Metrics (Accessed 2025-02-20).

[2] N. I. o. S. a. Technology, "Minimum Security Requirements for Federal Information and Information Systems," U.S. Department of Commerce, Tech. Rep., Mar. 2006. [Online]. Available: https://csrc.nist.gov/pubs/fips/200/final (Accessed 2025-04-15).

[3] N. Daswani and M. Elbayadi, *The Equifax Breach*. Apress, 2021, pp. 75–95. ISBN 978-1-4842-6655-7

[4] "Equifax data breach settlement." [Online]. Available: https://www.ftc.gov/enforcement/refunds/equifax-data-breach-settlement (Accessed 2024-09-02).

[5] L. Cheng, F. Liu, and D. D. Yao, "Enterprise data breach: causes, challenges, prevention, and future directions," *WIREs Data Mining and Knowledge Discovery*, vol. 7, no. 5, p. e1211, 2017.

[6] S. Seifermann *et al.*, "Detecting violations of access control and information flow policies in data flow diagrams," *Journal of Systems and Software*, vol. 184, p. 111138, 2022.

[7] M. Balliu, D. Schoepe, and A. Sabelfeld, "We are family: Relating information-flow trackers," in *Computer Security – ESORICS 2017*. Springer. ISBN 978-3-319-66402-6 pp. 124–145.

[8] A. Sabelfeld and A. Myers, "Language-based information-flow security," *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 1, pp. 5–19, 2003.

[9] H. Mantel, *Information Flow and Noninterference*. Springer, 2011, pp. 605–607. ISBN 978-1-4419-5906-5

[10] A. C. Myers, "Jflow: Practical mostly-static information flow control," in *Proc. of the 26th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*. ACM, 1999. ISBN 1581130953 p. 228–241.

[11] W. Enck *et al.*, "Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones," *ACM Trans. Comput. Syst.*, vol. 32, no. 2, 2014.

[12] S. Arzt *et al.*, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," in *Proc. of the 35th ACM SIGPLAN Conf. on Programming Language Design and Implementation*. ACM, 2014. doi: 10.1145/2594291.2594299. ISBN 9781450327848 p. 259–269.

[13] B. W. Boehm, R. K. Mcclean, and D. E. Urfrig, "Some experience with automated aids to the design of large-scale reliable software," *IEEE Trans. Softw. Eng.*, vol. SE-1, no. 1, pp. 125–133, 1975.

[14] S. Seifermann, "Architectural data flow analysis for detecting violations of confidentiality requirements," Ph.D. dissertation, Karlsruher Institut für Technologie (KIT), 2022.

[15] J. Epstein, S. Matsumoto, and G. McGraw, "Software security and soa: danger, will robinson!" *IEEE Security & Privacy*, vol. 4, no. 1, pp. 80–83, 2006.

[16] OWASP top ten | OWASP foundation. [Online]. Available: https://owasp.org/www-project-top-ten/ (Accessed 2023-07-18).

[17] J. Jürjens, "UMLsec: Extending UML for Secure Systems Development," in *UML 2002 — The Unified Modeling Language*, ser. Lecture Notes in Computer Science. Springer, 2002. ISBN 978-3-540-45800-5 pp. 412–425.

[18] J. Geismann, B. Haverkamp, and E. Bodden, "Ensuring threat-model assumptions by using static code analyses," in *ECSA 2021 Companion Volume*. CEUR-WS.org. [Online]. Available: http://ceur-ws.org/Vol-2978/mde4sa-paper1.pdf

[19] K. Tuma, R. Scandariato, and M. Balliu, "Flaws in flows: Unveiling design flaws via information flow analysis," in *2019 IEEE Int. Conf. on Softw. Archit.* doi: 10.1109/ICSA.2019.00028 pp. 191–200.

[20] T. Runge *et al.*, "Information flow control-by-construction for an object-oriented language," in *Softw. Eng. and Formal Methods*. Springer, 2022. ISBN 978-3-031-17108-6 pp. 209–226.

[21] M. E. Kramer *et al.*, "Model-Driven Specification and Analysis of Confidentiality in Component-Based Systems," Tech. Rep., 2017. [Online]. Available: http://dx.doi.org/10.5445/IR/1000076957

[22] M. Almorsy, J. Grundy, and A. S. Ibrahim, "Automated software architecture security risk analysis using formalized signatures," in *2013 35th Int. Conf. on Softw. Eng.* ISSN 1558-1225 pp. 662–671.

[23] D. Giffhorn and C. Hammer, "Precise analysis of java programs using joana," in *2008 8th IEEE Int. Working Conf. on Source Code Analysis and Manipulation*. IEEE, pp. 267–268.

[24] N. Broberg, B. van Delft, and D. Sands, "Paragon for practical programming with information-flow control," in *Programming Languages and Systems*. Springer, 2013. ISBN 978-3-319-03542-0 pp. 217–232.

[25] S. Greiner, "A framework for non-interference in component-based systems," Ph.D. dissertation, Karlsruhe Institute of Technology, 2018.

[26] B. Chess and J. West, *Secure programming with static analysis*. Pearson Education, 2007.

[27] A. Bucaioni *et al.*, "Architecture as code," in *22nd IEEE International Conference on Software Architecture*, December 2024. [Online]. Available: http://www.ipr.mdu.se/publications/7118-

[28] K. Tuma *et al.*, "Checking security compliance between models and code," *Software and Systems Modeling*, vol. 22, pp. 273–296, 2023.

[29] K. Katkalov *et al.*, "Model-driven development of information flow-secure systems with iflow," in *2013 Int. Conf. on Social Computing*. doi: 10.1109/SocialCom.2013.14 pp. 51–56.

[30] I. Ryan, U. Roedig, and K.-J. Stol, "Unhelpful assumptions in software security research," in *Proc. of the 2023 ACM SIGSAC Conf. on Computer and Communications Security*, ser. CCS '23. ACM, 2023. doi: 10.1145/3576915.3623122. ISBN 9798400700507 p. 3460–3474.

[31] S. Jasser, "Enforcing architectural security decisions," in *2020 IEEE Int. Conf. on Softw. Archit.* doi: 10.1109/ICSA47634.2020.00012 pp. 35–45.

[32] R. Heinrich, *Architecture-based Evolution of Dependable Software-intensive Systems*. KIT Scientific Publishing, 2023.

[33] R. Heldal *et al.*, "Descriptive vs prescriptive models in industry," in *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*, ser. MODELS '16. ACM. doi: 10.1145/2976767.2976808. ISBN 9781450343213 p. 216–226.

[34] C. Atkinson, D. Stoll, and P. Bostan, "Orthographic software modeling: A practical approach to view-based development," in *Evaluation of Novel Approaches to Software Engineering*. Springer, 2010. ISBN 978-3-642-14819-4 pp. 206–219.

[35] S. Hahner *et al.*, "Model-based confidentiality analysis under uncertainty," in *2023 IEEE 20th Int. Conf. on Softw. Archit. Companion*, pp. 256–263.

[36] M. Walter, R. Heinrich, and R. Reussner, "Architectural attack propagation analysis for identifying confidentiality issues," in *2022 IEEE 19th Int. Conf. on Softw. Archit.* doi: 10.1109/ICSA53651.2022.00009

[37] C. Gerking and D. Schubert, "Component-based refinement and verification of information-flow security policies for cyber-physical microservice architectures," in *2019 IEEE Int. Conf. on Softw. Archit.* doi: 10.1109/ICSA.2019.00015 pp. 61–70.

[38] A. Marback *et al.*, "A threat model-based approach to security testing," *Software: Practice and Experience*, vol. 43, no. 2, pp. 241–258, 2013.

[39] D. Xu *et al.*, "Automated security test generation with formal threat models," *IEEE Transactions on Dependable and Secure Computing*, vol. 9, no. 4, pp. 526–540, 2012.

[40] N. A. Almubairik and G. Wills, "Automated penetration testing based on a threat model," in *2016 11th Int. Conf. for Internet Technology and Secured Transactions (ICITST)*, 2016. doi: 10.1109/ICITST.2016.7856742 pp. 413–414.

[41] J. Dürrwang *et al.*, "Enhancement of automotive penetration testing with threat analyses results," *SAE Int. Journal of Transportation Cybersecurity and Privacy*, vol. 1, no. 11-01-02-0005, pp. 91–112, 2018.

[42] 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*. ACM, 2022. doi: 10.1145/3548606.3559394. ISBN 9781450394505 p. 7.

[43] K. Hermann *et al.*, "An exploratory study on the engineering of security features," 2025. [Online]. Available: https://arxiv.org/abs/2501.11546

[44] S. M. Peldszus, *The GRaViTY Framework*. Springer, 2022, pp. 383–392. ISBN 978-3-658-37665-9

[45] M. Eby *et al.*, "Integrating security modeling into embedded system design," in *14th Annu. IEEE Int. Conf. and Workshops on the Eng. of Computer-Based Systems*. doi: 10.1109/ECBS.2007.45 pp. 221–228.

[46] J.-P. Töberg *et al.*, "Modeling and enforcing access control policies for smart contracts," in *2022 IEEE Int. Conf. on Decentralized Applications and Infrastructures (DAPPS)*, pp. 38–47.

[47] 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*. doi: 10.1109/DS-RT.2013.8 pp. 3–9.

[48] C. Talcott *et al.*, *Composition of Languages, Models, and Analyses*. Springer, 2021, pp. 45–70. ISBN 978-3-030-81915-6

[49] H. Stachowiak, *Allgemeine Modelltheorie*. Springer, 1973. ISBN 0387811060

[50] T. Stahl and M. Völter, *Model-driven software development : technology, Eng. , management*. Wiley, 2006. ISBN 0470025700; 9780470025703

[51] M. Brambilla, J. Cabot, and M. Wimmer, "Model-driven software engineering in practice," *Synthesis Lectures on Soft. Eng.*, vol. 3, no. 1, pp. 1–207, 2017.

[52] A. Schürr and F. Klar, "15 years of triple graph grammars," in *Graph Transformations*. Springer, 2008. ISBN 978-3-540-87405-8 pp. 411–425.

[53] R. Heinrich, "Architectural runtime models for integrating runtime observations and component-based models," *Journal of Systems and Software*, vol. 169, p. 110722, 2020.

[54] C. Talcott *et al.*, *Foundations*. Springer, 2021, pp. 9–37. ISBN 978-3-030-81915-6

[55] R. Heinrich, M. Strittmatter, and R. H. Reussner, "A layered reference architecture for metamodels to tailor quality modeling and analysis," *IEEE Trans. Softw. Eng.*, 2019.

[56] D. Bell, "Looking back at the bell-la padula model," in *21st Annu. Computer Security Applications Conf. (ACSAC'05)*. doi: 10.1109/CSAC.2005.37 pp. 15 pp.–351.

[57] D. E. Denning, "A lattice model of secure information flow," *Commun. ACM*, vol. 19, no. 5, p. 236–243, 1976.

[58] F. B. Schneider, "Enforceable security policies," *ACM Trans. Inf. Syst. Secur.*, vol. 3, no. 1, p. 30–50, 2000.

[59] D. Helm *et al.*, "Modular collaborative program analysis in opal," in *Proc. of the 28th ACM Joint Meeting on European Softw. Eng. Conf. and Symp. on the Foundations of Softw. Eng.*, ser. ESEC/FSE 2020. ACM, 2020. ISBN 9781450370431 p. 184–196.

[60] D. Harel and B. Rumpe, "Meaningful modeling: what's the semantics of "semantics"?" *Computer*, vol. 37, no. 10, pp. 64–72, 2004.

[61] M. Konersmann *et al.*, "Towards a semantically useful definition of conformance with a reference model," *Journal of Object Technology*, vol. 23, no. 3, pp. 1–14, 2024, the 20th European Conference on Modelling Foundations and Applications (ECMFA 2024).

[62] R. H. Reussner *et al.*, *Modeling and Simulating Software Architectures – The Palladio Approach*. MIT Press, 2016. ISBN 9780262034760

[63] N. Boltz *et al.*, "An extensible framework for architecture-based data flow analysis for information security," in *Software Architecture. ECSA 2023 Tracks, Workshops, and Doctoral Symp.* Springer. ISBN 978-3-031-66326-0 pp. 342–358.

[64] S. G. Koch, "A reference structure for modular model-based analyses," Ph.D. dissertation, Karlsruher Institut für Technologie (KIT), 2023.

[65] CodeQL. [Online]. Available: https://codeql.github.com/ (Accessed 2024-09-02).

[66] 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, Tech. Rep., 2014, (Accessed 2024-01-08).

[67] A. Bucaioni *et al.*, "Continuous conformance of software architectures," in *2024 IEEE 21st Int. Conf. on Softw. Archit.* doi: 10.1109/ICSA59870.2024.00019 pp. 112–122.

[68] F. Reiche, R. Heinrich, and R. Reussner, "Supplementary material of publication detecting information flow security vulnerabilities by analysis coupling," Apr. 2025. [Online]. Available: https://doi.org/10.5281/zenodo.15263722

[69] H. Klare *et al.*, "Enabling consistency in view-based system development — the vitruvius approach," *Journal of Systems and Software*, vol. 171, p. 110815, 2021.

[70] S. Becker, "Coupled model transformations for qos enabled component-based software design," Ph.D. dissertation, Universität Oldenburg, 2008.

[71] F. Reiche *et al.*, "Consistency management for security annotations for continuous verification," in *ACM / IEEE 27th Int. Conf. on Model Driven Engineering Languages and Systems Companion*, 2024.

[72] F. Reiche, R. Heinrich, and R. Reussner, "Replication package of publication detecting information flow security vulnerabilities by analysis coupling," Apr. 2025. [Online]. Available: https://doi.org/10.5281/zenodo.15266366

[73] N. Polikarpova *et al.*, "What good are strong specifications?" in *2013 35th International Conference on Software Engineering (ICSE)*, 2013. doi: 10.1109/ICSE.2013.6606572 pp. 262–271.

[74] "SpecificationTransfer." [Online]. Available: https://github.com/KASTEL-CSSDA/SpecificationTransfer (Accessed 2025-02-03).

[75] "Architecture-And-StaticCode-Analyses-CouplingFramework." [Online]. Available: https://github.com/KASTEL-CSSDA/Architecture-And-StaticCode-Analyses-CouplingFramework (Accessed 2025-02-03).

[76] V. Basili, G. Caldiera, and H. D. Rombach, "The goal question metric approach," in *Encyclopedia of Softw. Eng.*, 2nd ed. John Wiley & Sons, Inc., 2002, pp. 1–10. ISBN 0471028959

[77] A. Kaplan *et al.*, "Introducing an evaluation method for taxonomies," in *2022 Int. Conf. on Evaluation and Assessment in Softw. Eng.* ACM, 2022. doi: 10.1145/3530019.3535305. ISBN 978-1-4503-9613-4 pp. 311–316.

[78] D. C. Muñoz Hurtado, S. Serbout, and C. Pautasso, "Mining security documentation practices in openapi descriptions," in *2025 IEEE 22nd Int. Conf. on Softw. Architect.* doi: 10.1109/ICSA65012.2025.00021 pp. 119–130.

[79] K. Katkalov *et al.*, "Evaluation of jif and joana as information flow analyzers in a model-driven approach," in *Data Privacy Management and Autonomous Spontaneous Security*. Springer, 2013. ISBN 978-3-642-35890-6 pp. 174–186.

[80] A. van den Berghe *et al.*, "Design notations for secure software: a systematic literature review," *Software & Systems Modeling*, vol. 16, no. 3, 2017.
[81] D. Steinberg, *EMF - Eclipse modeling framework* /, 2nd ed., ser. The eclipse series. Addison-Wesley,, 2009.
[82] Modeling the travel planner application with IFlow. [Online]. Available: https://kiv.isse.de/projects/iflow/TravelPlannerSite/index.html (Accessed 2024-09-02).
[83] "Eclipse Secure Storage." [Online]. Available: https://help.eclipse.org/latest/topic/org.eclipse.platform.doc.isv/guide/secure_storage.htm?cp=2_0_3_7_0 (Accessed 2025-03-04).
[84] S. Herold *et al.*, *CoCoME - The Common Component Modeling Example*. Springer, 2008, pp. 16–53. ISBN 978-3-540-85289-6
[85] S. Greiner and M. Herda, "Cocome with security," Karlsruher Institut für Technologie (KIT), Tech. Rep. 2, 2017.
[86] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empirical Softw. Eng.*, vol. 14, no. 2, pp. 131–164, 2009.
[87] P. Muntean *et al.*, "Automated detection of information flow vulnerabilities in uml state charts and c code," in *2015 IEEE Int. Conf. on Software Quality, Reliability and Security - Companion*. doi: 10.1109/QRS-C.2015.30 pp. 128–137.
[88] K. Yurchenko *et al.*, "Architecture-driven reduction of specification overhead for verifying confidentiality in component-based software systems." in *MODELS (Satellite Events)*, 2017, pp. 321–323.
[89] P. Meier, S. Kounev, and H. Koziolek, "Automated transformation of component-based software architecture models to queueing petri nets," in *2011 IEEE 19th Annu. Int. Symp. on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*. doi: 10.1109/MASCOTS.2011.23 pp. 339–348.
[90] S. Hahner *et al.*, "Modeling data flow constraints for design-time confidentiality analyses," in *2021 IEEE 18th Int. Conf. Softw. Archit. Companion*. doi: 10.1109/ICSA-C52384.2021.00009 pp. 15–21.
[91] B. Combemale, L. Gonnord, and V. Rusu, "A generic tool for tracing executions back to a dsml's operational semantics," in *Modelling Foundations and Applications*. Springer, 2011. ISBN 978-3-642-21470-7 pp. 35–51.
[92] V. Cortellessa, R. Eramo, and M. Tucci, "From software architecture to analysis models and back: Model-driven refactoring aimed at availability improvement," *Information and Software Technology*, vol. 127, p. 106362, 2020.
[93] D. Monschein *et al.*, "Enabling consistency between software artefacts for software adaption and evolution," in *2021 IEEE 18th Int. Conf. on Softw. Archit.* doi: 10.1109/ICSA51549.2021.00009
[94] M. B. Dwyer and S. Elbaum, "Unifying verification and validation techniques: relating behavior and properties through partial evidence," in *Proc. of the FSE/SDP Workshop on Future of Softw. Eng. Research*. ACM, 2010. ISBN 9781450304276 p. 93–98.
[95] S. Cruanes *et al.*, "Tool integration with the evidential tool bus," in *Verification, Model Checking, and Abstract Interpretation*. Springer, 2013. ISBN 978-3-642-35873-9 pp. 275–294.

Frederik Reiche is a doctoral researcher in the Dependability of Software-intensive Systems group at Karlsruhe Institute of Technology (KIT). He holds an M.Sc. degree in Computer Science from KIT. His primary research interests cover model-based analysis composability, model-driven engineering, and software security analysis. His research is mainly located in the quality analysis of software systems.

Ralf Reussner is full professor for software engineering at Karlsruhe Institute of Technology (KIT) since 2006. He leads the group for Dependability of Software-intensive Systems and heads the KASTEL-Institute for Information Security and Dependability. His research group works in the interplay of software architecture and predictable software quality as well as on view-based design methods for software-intensive technical systems.

Robert Heinrich is full professor for software engineering at Ulm University since 2025. He leads the group for Quality-driven System Evolution. Before he was a research group leader at KIT. His research interests include software quality modeling and analysis, in particular, the (de)composition of model-based analysis to provide flexibility in model-driven engineering.