Detecting Encryption Vulnerabilities By Coupling Architectural Analyses and Source Code Analyses

Frederik Reiche
Karlsruhe Institute of Technology (KIT)
Karlsruhe, Germany
frederik.reiche@kit.edu

Robert Heinrich
Karlsruhe Institute of Technology (KIT)
Karlsruhe, Germany
robert.heinrich@kit.edu

***Abstract*—Architectural security analyses calculate security vulnerabilities by evaluating architectural security design models comprising the system architecture and security-related information. The architectural analysis is performed before the implementation phase to avoid implementing a vulnerable system. Consequentially, the architectural vulnerabilities are calculated based on the assumption that the implementation complies with the specified system. When the implementation does not comply with the security design models, the architectural analysis may miss vulnerabilities in the final system. We address this problem by presenting an approach for analysis coupling, which allows the architectural analysis to be performed with information about security weaknesses regarding data encryption in the implementation detected by a source code analysis searching for predefined patterns. We perform a case study-based evaluation of the accuracy to detect architectural vulnerabilities arising from weaknesses in the implementation. In this evaluation, we apply the coupling approach to couple an architectural analysis with three source code analyses and apply them to three systems containing encryption-related weaknesses. Our evaluation shows that the coupling enables the detection of architectural vulnerabilities that are not detectable when performing the architectural analysis in isolation. However, the evaluation shows negative impacts of our coupling approach by missing existing vulnerabilities or reporting not existing vulnerabilities.**

***Index Terms*—Model-driven Security, Analysis Coupling, Software Architecture Analysis, Source Code Analysis**

I. Introduction

The rising complexity of software systems makes it challenging to ensure security objectives like the integrity or confidentiality of data [1]. Due to this complexity, security vulnerabilities are introduced in the software system, allowing attackers to alter, disrupt, or destroy sensitive information [2].

Consequently, security vulnerabilities must be detected and fixed before being exploited. Static security analysis is an established method for detecting security vulnerabilities in source code without deploying the system. However, around 50% of vulnerabilities originate from flaws in the security design [3]. Based on these insights, architectural security analyses like [4]–[6] emerged in the last decades, investigating security vulnerabilities before system implementation. These analyses use an architectural security design model comprising the architectural design and security specifications defining security-related properties for architectural elements, called security characteristics [4]. However, the architectural security design is an abstraction of the implemented system and omits details. For example, SecDFD [7] specifies data encryption while omitting details about the encryption algorithms. This requires *software engineers* to perform the implementation. This is also true for software evolution because the implementation should be performed when the design is free of vulnerabilities to avoid implementing an already vulnerable system. Therefore, the architectural analysis calculates the architectural vulnerabilities by assuming the implementation to realize the architectural system design [7], [8]. However, at some point in software evolution, the implementation may not comply anymore with the architectural security design [7]. A non-compliant implementation regarding specified security characteristics in the architectural security design can result in the architectural analysis not detecting vulnerabilities in the final system. Approaches detecting non-compliance exist, such as [7], [9], [10]. However, these approaches leave the interpretation of the non-compliance on the architectural security up to the *software engineers*. Because non-compliances arise from challenges in realizing architectural security design in the implementation, interpreting the effect of non-compliance on architectural security can be assumed equally challenging. We address this challenge by presenting an approach for coupling architectural analyses and static source code security analyses. In this coupling, the architectural analysis uses information about weaknesses in the implementation, obtained from source code analyses. With this approach, architectural analysis can detect vulnerabilities that are not detectable when performed in isolation because it considers the information from the implementation rather than depending on assumptions made by the software architect in the design phase. We want to make clear that this approach does not contradict the understanding that architectural analysis must be performed before implementing the system. Still, our coupling can provide insights into the impact of non-compliance on architectural security.

Challenges in data encryption are listed among the top security risks by OWASP Top 10 [11]. This is why several architectural analyses specify encrypted data communication, e.g., in [4]–[6]. Also, several source code analyses detect weaknesses related to data encryption by searching for predefined patterns in the implementation, such as [12]–[14]. We call source code analyses searching for patterns that indicate weaknesses *pattern-search-based*. Detecting weaknesses is relevant because they are a prerequisite for a vulnera-

bility to occur [15]. Due to the importance of this topic, we describe our analysis coupling for architectural analyses specifying security characteristics concerning encryption and pattern-search-based source code analyses detecting patterns that indicate weaknesses impacting data encryption. In the following, we call these analyses *architectural analyses* and *source code analyses*. We state the research question **RQ: How can we enable the coupling of architectural analysis and pattern-search-based source code analyses to detect the impact of weaknesses on the architectural security regarding encryption**. We answer this question by providing a coupling approach comprising two contributions:

- **C1** We describe how the architectural and pattern-search-based source code analysis interact in the coupling approach and which artifacts of the analyses are used to enable the coupling.
- **C2** A coupling can only be performed if the information in the analyses can be connected. However, the architectural analysis and source code analysis provide different but similar information, such as the security characteristics and weaknesses violating them. For this, our coupling approach defines artifacts that enable bridging this gap to enable the coupling.

The remainder of this paper is structured as follows: Section II presents a running example to illustrate our approach. Section III provides necessary foundations for this paper. The coupling design, i.e., which artifacts of the analyses are used in our coupling and the process for exchanging information is described in Section IV. Based on this design, Section V presents how the information exchange is enabled for coupling architectural analyses and pattern-search-based source code analyses. In Section VI, we present the evaluation of our coupling approach. In Section VII, we discuss limitations of our coupling and future work to address them. Related work is discussed in Section VIII and Section IX concludes this paper.

II. Running Example

Our running example is illustrated in Figure 1. It comprises (a) an example system represented by an architectural security design and an implementation comprising encryption weaknesses, (b) an example architectural analysis, and (c) an example source code analysis.

System: We adapt the *JPMail* system used by Seifermann et al. [16]. The software architecture is shown on the left-hand side of Figure 1. Three components exist: *MailSendingClient*, *MailService* and *MailReceivingClient*, which are deployed on the resources *AlicePC*, *MailServer* and *BobPC*, respectively. The system communicates an email from *MailSendingClient* to *MailReceivingClient* over *MailService*. The resources are connected by two communication links. The data communicated over these links must be encrypted to prevent potential attackers from accessing the email content. The *MailSendingClient* provides a service *sendMail* receiving the email *header* and email *body* as parameters from the user. The *MailService* and *MailReceivingClient* both provide the service *dispatchMail* receiving the *header* and *body* as Strings representing the encrypted data. When *dispatchMail* of the *MailService* is called, it routes the encrypted *header* and *body* to the *MailReceivingClient*. The implementation of the *sendMail* service of the *MailSendingClient* component is shown on the right-hand side of Figure 1. In the *sendMail* method, the *Cipher cipher* is generated using the encryption algorithm *DES* to perform the encryption. The method *encrypt* is called to encrypt the *header* and *body*, resulting in *encHeader* and *encBody*, which are sent to the *MailService* by calling the *dispatch* method.

Architectural Analysis: We use an architectural analysis similar to *Secure Links* of UMLSec [6]. UMLSec specifies the requirement *secrecy* for communications between. components. The security characteristic *encryption* defines encrypted communication over the communication links. In *UMLSec*, the security characteristic *encryption* specifies that encryption is performed on the physical layer. To illustrate capabilities in the coupling, we define the *encryption* on communication links to assume some form of encryption, i.e., either encryption of data or using an encrypted protocol such as *HTTPS*. The analysis detects a vulnerability when *secrecy* is specified but a corresponding communication link is not encrypted.

Source Code Analysis: We use the *source code analysis* FindSecBugs [13] to detect the usage of broken cryptographic algorithms. For this, the analysis searches for patterns in the implementation, like usage of the Java Cryptographic API with the cryptographic algorithm *DES*. The source code analysis finds the pattern *DES_USAGE* at the *Cipher.getInstance* call and reports its location as a combination of the java class, the java method, and the line of code in which it occurs.

III. Background

Our coupling approach uses model-driven technologies to couple model-based security analyses. In this section, we introduce the necessary background for this paper.

A. Models, Metamodels and Transformations

We realize the coupling betweenthe input and output models of the analyses. Models are abstractions of a real or virtual entity for a given purpose with a subset of its attributes [17]. We also comprehend the implementation as a model, as it does the system behavior while committing information like deployment. The structure of a *model* is described by a *metamodel*, comprising *metaclasses* and *references* between metaclasses describing directed relations [18]. A *Model Transformation* transforms elements of one or more input models to corresponding elements of one or more output models [19].

B. Model-Based Analysis

Our coupling approach connects *model-based security analyses*. 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* [20]. A *security analysis* typically answers the *question* of whether a specific weakness or vulnerability exists in a given system. In our coupling approach, we focus on analyses using metamodels and models [18] for describing their input and output.

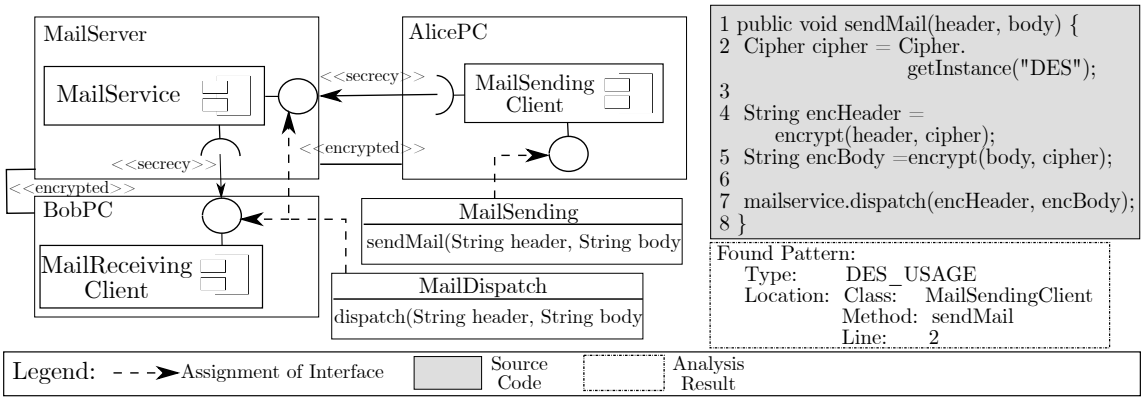

Fig. 1. Combined diagrams showing parts of the architectural security model (left), the implementation of the sendMail Method from the MailSendingClient (upper right), and the report of the source code analysis (lower right). Adapted from Seifermann et al. [16]. Architectural elements based on UMLSec [6]

C. Source Code Security Analysis

Source code security analyses can be differentiated into property analyses and bug-searching analyses [21]. Bug-searching analyses search for patterns related to weaknesses in the source code. This is relevant as a weakness is a precondition for a vulnerability to occur [15]. A vulnerability can be exploited to violate a *security objective* of the system, such as protecting data *confidentiality*. In this paper, we call them *pattern-search-based analyses* to highlight the usage of these patterns. Property analyses do not search for implementation weaknesses but use a specification to evaluate whether a security property like information flow security holds.

IV. Coupling Design

A coupling design comprises a *coupling form* and a *coupling process*. The *coupling form* defines the artifacts used to establish the coupling [20]. VBased on the selected *coupling form*, a *coupling process* defines how the analyses interact to achieve the coupling [22]. We use *roles* in the analysis coupling to clearly separate responsibilities in creating coupled analyses.

A. Selecting the Coupling Form

Architectural analyses and source code analyses are designed for independent application in different stages of software development. In a development process, architectural security vulnerabilities detected by an architectural analysis must be fixed before the implementation to avoid realizing a system architecture already known to be vulnerable. Coupling approaches that combine the metamodels of the analyses (white-box) or integrate functionality for information exchange into the coupling tools (grey-box) requires maintaining two versions of the analyses: one that can perform the architectural analysis in isolation and one that can perform the architectural analysis in the coupling. We select a *black-box coupling form* using the input and output models of the analyses and transformations between them to achieve the coupling. The black-box coupling allows the application of the architectural analysis in isolation and the coupling without further modification because the transformations can be independently executed.

B. Coupling Process

The *coupling process* defines how the analyses interact to achieve the coupling goal. The architectural security design should only be implemented when the architectural analysis does not report any vulnerabilities, i.e., the output of the architectural analysis is empty. Coupling processes combining the results of analyses require knowledge about how architectural vulnerabilities are calculated to add new ones. We use a *sequential coupling* where the architectural analysis is *parameterized* by modifying its input model with the information from the source code analysis output model.

The output of the source code analyses comprises binary statements of whether a pattern indicating a weakness is detected. Consequently, the coupling can only maintain or remove specified security characteristics in the architectural model that are violated by weaknesses. Describing security characteristics for architectural elements can be performed by annotation mechanisms, (meta)models or as attributes of architectural elements [23]. We use the term annotation for all of these approaches because the usage of attributes can be restructured into annotations [23]. The coupling is performed by interpreting the output model of the source code analyses to decide whether a security characteristic specified for an architectural element in the input model of the architectural analysis is violated. If a violation occurs, the annotation of the security characteristic is removed. The detection of the usage of the weak encryption algorithm *DES* in the running example is reported in the source code analysis output, which threatens the *encryption* specified in the architectural security design. Thus, we parameterize the architectural analysis by removing the annotation specifying the security characteristic *encrypted*. This coupling approach is highly dependent on the granularity of the annotated architectural elements. In the running example, the coupling removes the specified *encryption*. Consequently, the architectural analysis interprets every data transferred over the communication link as unencrypted. In contrast, removing an annotation from a single action, like a service call, affects only the data associated with this action. We discuss this limitation further in Section VII.

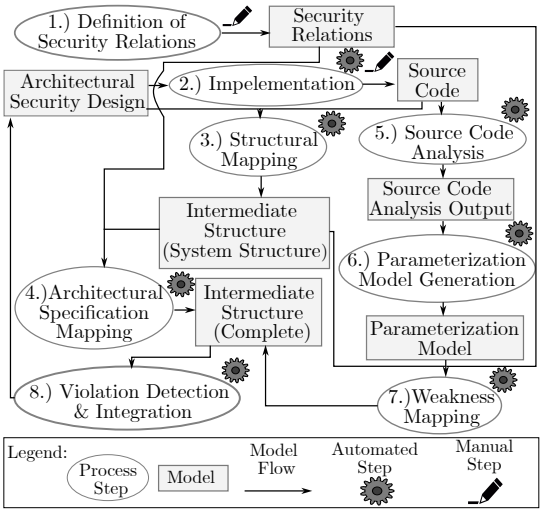

Fig. 2. Models and process steps in the coupling approach. Sequential increasing numbers order the execution of the coupling process steps.

C. Roles in the Analysis Coupling

For this coupling approach, we distinguish two roles. The *analysis architect*, described by Koch [24], creates an analysis by assembling analysis components. This *analysis architect* is an expert in the respective abstraction level of its analysis and knows the meaning of the element in the input and output (meta)models of the analyses. For instance, the *analysis architect* of the source code analysis knows the relation between a pattern detected by the source code analysis and an associated security weakness. We introduce a *coupling expert*, which defines the transformations in the coupling. The primary competence of the *coupling expert* is determining relations between the elements of the input and output (meta)models of the architectural analysis and source code analysis.

V. Coupling Approach

Our coupling approach realizes the sequential black-box coupling performing several process steps as illustrated in Figure 2 In this process, we call the input model of the architectural analysis the *Architectural Security Design*, the input model of the source code analysis *Source Code*, and the output of the source code analysis *Source Code Analysis Result*. We omit the architectural analysis step as it is not involved in establishing the coupling. Each process step can either be *automated*, e.g., by automated model transformation or must be performed *manually*. *Automated* process steps are beneficial as they introduce a one-time effort for their creation and can be reused for arbitrary systems afterwards. *Manual* process steps require effort for every execution of the coupling.

The coupling faces the challenge of a *Architectural Security Design* and a *Source Code Analysis Result* comprising different but related information. The *Architectural Security Design* comprises security characteristics, like *encryption* in the running example. In contrast, the *Source Code Analysis Result* comprises patterns related to *weaknesses* violating the security characteristic, such as *weak encryption*. *Architectural*

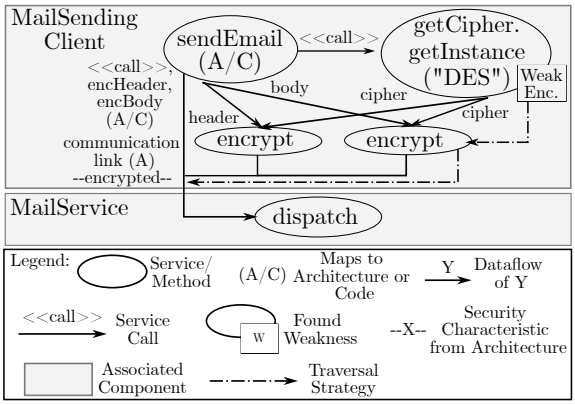

Fig. 3. Intermediate structure for the first part of running example. Only elements mapping to the architectural model are annotated with (A or C).

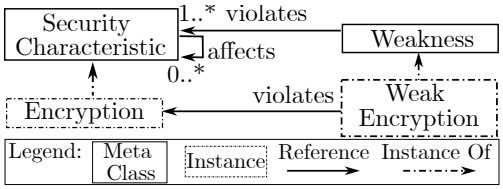

Fig. 4. The Security Relations metamodel for relating Security Characteristics and Weaknesses and its instance for the running example

Security Design models can specify security characteristics for architectural elements describing information transfer, like communication links in UMLSec [6]. In contrast, source code analyses can report locations in the code, such as a statement calling another method. We address these challenges using an intermediate structure to unify the system structure, the specified security characteristics and the detected weaknesses.

A black-box coupling implies that the *Source Code Analysis Result* must comprise enough information to identify the violated security characteristic specified in the *Architectural Security Design*. We found that the *Source Code Analysis Result* can comprise information that must be interpreted to relate it to the *Architectural Security Design*, e.g., the detected pattern is provided rather than an associated weakness. This makes it more difficult for a *coupling expert* to create the transformations. We address this challenge with a *Parameterization Metamodel* as an abstraction of the *Source Code Analysis Result*, capturing all information from the intermediate structure necessary to perform the coupling.

A. The Intermediate Structure

The coupling approach determines which annotation in the *Architectural Security Design* is affected by a weakness by using an intermediate structure to capture correspondences between (1) elements in the *Architectural Security Design* and the *Source Code* and (2) *weaknesses* found in the *Source Code* and *security characteristics*. Figure 3 illustrates an exemplary intermediate structure for the running example.

a) Capturing Relations between Security Characteristics and Weaknesses: The *Source Code Analysis Result* comprises

patterns indicating *Weaknesses* in the coupling. These *Weaknesses* violate a specified *Security characteristic*. We capture this relation in a *Security Relations* metamodel comprising the metaclasses *Security Characteristic* and *Weakness* as illustrated in Figure 4. A relation *violates* expresses that a *Weakness* results in a violation of one or more *Security Characteristics*. The *Security Characteristics* describe security concepts like encryption or input sanitation in [6], [10]. A similar relation is provided by the Common Weakness Enumeration (CWE) [25] with the distinction that these weaknesses are related to high-level security objectives such as confidentiality. A *coupling expert* must either create a *Security Relations* model manually in the *Definition of Security Relations* step (step 1) or reuse an existing one. The model in the running example comprises the *Security Characteristic encryption* and the *Weakness weak encryption* violating it.

We want to clarify that the elements *Security Characteristics* and *Weaknesses* are not metamodel elements of specific analyses but capture a concept like *encryption*. Thus, this design requires transformations from the specific model elements in the *Architectural Security Design* and the source code analysis output model to *Security Characteristics* and *Weaknesses*.

b) Capturing Architectural Elements and Implementation Elements: We assume the *Source Code* resulting from the *Implementation* step (step 2) to be consistent with the *Architectural Design Model*. This implementation can be performed semi-automated, e.g., by generation techniques [18]. Still, *software engineers* must complete the implementation manually as described in Section I.

For the coupling, the architectural elements and implementation elements describing the system must be related in addition to relating the security characteristics and the weaknesses. The approach in this paper requires relating model elements with different but related concepts. In our running example, the *call* of *dispatch* defines an action introducing a data flow to the calling method. In contrast, the link between the components defines a communication channel through which data flows. Therefore, the intermediate structure must comprise elements to which the call and the communication link can be mapped. For the encryption vulnerabilities in this paper, the intermediate structure has to capture (a) services in the architecture and corresponding methods in the implementation, (b) internal methods of the implementation, and (c) communication between the information in (a) and (b) as illustrated in Figure 3. Examples of structures are call graphs or the GRaViTY program model [26]. The *coupling expert* defines two transformations for the *Structural Mapping* (step 3) to create the intermediate structure: one transformation to map the structural elements of the *Architectural Security Design* to the intermediate structure and one to map the structural elements of the *Source Code* to the intermediate structure (step 3). This results in the benefit that transformations can be reused when creating new couplings with analyses of a system description language in an existing coupling.

In step 3, the *coupling expert* has to ensure that the intermediate structure captures the architectural system design and the implementation consistently, i.e., the mapping of the services in the architectural model and the methods in the implementation have to match, as well as the created calls and communication link relations. In our running example, when the method *sendMail* in the implementation is mapped to an element in the intermediate structure, a mapping for the service *sendMail* in the architectural model must exist. When the intermediate structure captures the call of the method *dispatch* in the implementation, the corresponding communication link over which this communication is established must also be mapped to the intermediate structure. An instance of an exemplary intermediate structure for *sendMail* is shown in Figure 3. The nodes *sendMail* and *dispatch* map to the services in the architectural model and the methods in the implementation. The remaining nodes *getCipher* and *encrypt* only map to methods in the implementation. The edges represent either a call or data flow, e.g., *sendMail* calls *dispatch*, and a data flow exists of *encHeader* to *dispatch*. The edges to *dispatch* capture a mapping to the communication link.

c) Attaching Security Characteristics and Weaknesses to the Intermediate Structure: The *Security Characteristics* and *Weaknesses* in a *Security Relation* model must be representable in the intermediate structure to be usable by the coupling approach, e.g., to capture them for communication relations and the involved services. The *coupling expert* has to perform an *automated Architectural Specification Mapping* (step 4), transforming the security characteristics specified in the *Architectural Security Design* into the intermediate structure for the corresponding intermediate structure elements. This transformation must be created once for a single architectural analysis and intermediate structure. In our running example, the *encryption* security characteristic is provided to all edges mapped to the communication link. The *coupling expert* has to interpret the *Source Code Analysis Result* to extract the weaknesses for the implementation elements corresponding to the intermediate structure. We support this task with our *Parameterization Metamodel*.

B. Parameterization Metamodel

We found two challenges for the coupling, originating from the output of pattern-search-based *Source Code Analysis* (step 5): First, the *Source Code Analysis Result* often comprises the implementation elements involved in the patterns by describing the location, e.g., through lines of codes as illustrated in Figure 1. This format allows to navigate the source code precisely but requires the *coupling expert* to interpret the locations for resolving architectural elements. Secondly, the result may comprise the detected pattern rather than the weakness associated with it. The missing report of an associated weakness requires the *coupling expert* to understand the patterns to relate them to security characteristics.

We address these challenges by introducing the *Parameterization Metamodel*. The metamodel comprises an *ResultEntry* metaclass, abstracting a single report of a detected pattern. The *ResultEntry* references one element of the intermediate structure used in the coupling, e.g., a node representing a

service or method in Figure 3 to represent the affected element of the implementation usable for the coupling. A *ResultEntry* also comprises a *Weakness* to identify a security characteristic violated by the pattern represented in the *Source Code Analysis Result*. In the *Parameterization Model Generation* (step 6), an automated transformation from the *Source Code Analysis Result* to a *Parameterization Model* can be applied. The transformation must transform information about the location of the detected pattern into the intermediate structure. This transformation is created by the *analysis architect* supported by the *coupling expert*. The *analysis architect* can relate the implementation element in the analyzed source code to reported lines of code. The *analysis architect* defines a transformation from the representation of the implementation element, e.g., lines of code, to the actual implementation element. The *coupling expert* defines a transformation from the implementation element to the element of the intermediate structure. Consequently, the *coupling expert* is concerned with the intermediate structure to bridge the gap between architecture and implementation, while the *analysis architect* is concerned solely with the source code domain. We assume the relation between pattern and weakness to be known by the *analysis architect* rather than the *coupling expert*. This is why the *analysis architect* defines a transformation providing a *Weakness* for *patterns* detectable by the source code analysis. The *Parameterization Model* is used in the automated *Weakness Mapping* (step 7) to map the *Weaknesses* in each *ResultEntry* to the element of the intermediate structure. In our running example, the weakness *Weak Encryption* is mapped to the node *Ciper.getInstance("DES")*.

C. Realizing the Coupling

We realize the coupling with the intermediate structure by an *automated* transformation in the *Violation Detection & Integration* step (step 8), performing two activities: First, the coupling performs a *Violation Detection* to determine which security characteristic in the intermediate structure is violated by a reported weakness. Secondly, the coupling performs an *Integration* of the detected violation by removing the annotation in the *Architectural Security Design* model corresponding to the security characteristic in the intermediate structure.

We address the first activity using *automated search strategies* defined by a *coupling expert* for *Weaknesses* and *violated Security Characteristics*. These *search strategies* define how the intermediate structure is traversed to determine which specified security characteristic is violated by a weakness. These strategies are specific to the intermediate structure. Consequentially, the *coupling expert* has to define a *search strategy* once for every instance pair of *Weakness* and *Security Characteristic* with a *violates* relationship and an intermediate structure. A *search strategy* uses the intermediate structure element annotated comprising a weakness as the origin. In this paper, we focus on *search strategies* detecting whether a weakness associated with a method can be related to a security characteristic specified for a communication representation. The intermediate structure is traversed until the *next* element representing a communication with a mapping to the *Architectural Security Design* is reached, as illustrated in Figure 3. If the element representing a communication is annotated with a *Security Characteristic* violated by the *Weakness*, the annotation of the corresponding security characteristic in the *Architectural Security Design* model is removed. For this removal, the *coupling expert* provides an *automated* transformation, which resolves the annotation to be removed by searching the architectural elements corresponding to the intermediate structure element representing the communication and the corresponding *Security Characteristic*. The traversal is stopped at the first element representing a communication, mapped to the *Architectural Security Design* model. This automated transformation must be defined once for every *Security Characteristic* in the *Security Relation* model and corresponding specific security characteristic of an architectural analysis.

This approach cannot cover all specified security characteristics affected by a weakness, e.g., when the data propagates to different communication links. The specified security characteristic *encrypted* in the running example should be removed for both communication links because the weakly encrypted data is communicated through both. However, we made this design decision deliberately to consider only the next communication link for two reasons: First, a comprehensive further traversal requires a complete information flow analysis, which is infeasible to replicate in the coupling approach. Secondly, we only establish the relation based on the weaknesses reported by the source code analysis and the annotated security characteristics. Without further information, the *search strategy* cannot determine whether the weakness is still valid for consecutive communication links. If the *MailService* would decrypt the *encBody* and encrypt it again with a strong encryption algorithm, the *encryption* specified for the communication link between *MailServer* and *BobPC* is still valid. Removing it would result in false reports by the coupling. We discuss future work to address this limitation in Section VII.

VI. Evaluation

We guide the evaluation with the *Goal-Question-Metric* approach of Basili et al. [27]. We first discuss the *Goals*, *Questions* and *Metrics*. We then present the study design, the evaluation results and a discussion of threats to validity.

A. Goals, Questions and Metrics

We evaluate the accuracy of the analysis coupling regarding architectural vulnerabilities resulting through implementation weaknesses with **G1: Accuracy**. For this goal, we ask two questions: **Q1** Can we detect architectural vulnerabilities with our coupling approach originating from weaknesses in the implementation? **Q2** Does the coupling approach report vulnerabilities that do not exist or miss vulnerabilities? We answer **Q1** and **Q2** by inserting weaknesses violating data encryption in the system implementations that violate security characteristics in the architectural design, leading to architectural vulnerabilities. Related work such as [16], [26] also injects weaknesses to

evaluate accuracy. For answering **Q1** and **Q2**, we use the metrics *precision* p and *recall* r, commonly used by related work such as [26], [28], [29] for assessing the accuracy of analyses regarding known vulnerabilities. Precision $p = {t_p}/{(t_p + f_p)}$ and recall $r = {t_p}/{(t_p + f_n)}$ are calculated by considering true positives t_p, false positives f_p, and false negatives f_n of the expected results of the architectural analysis. We count a true positive if the architectural analysis reports a vulnerability for communicated data affected by encryption weaknesses. We count a false positive if the architectural analysis reports a vulnerability for communicated data unaffected by encryption weaknesses. We count a false negative if the architectural analysis reports no vulnerability despite being expected due to communicated data being affected by encryption weaknesses.

The architectural analysis in isolation cannot find the designed architectural vulnerabilities because it does not consider the information from the implementation. Still, true positives indicate vulnerabilities that would be missed by the architectural analysis performed in isolation. False positives indicate incorrect reports of architectural vulnerabilities, which cannot be reported by the architectural analysis performed in isolation.

B. Study Design

We use a case study-based design to calculate the metrics *precision* and *recall*. Case studies are often used in related work for evaluating architectural analyses, e.g., [5], [6], [16], [28]. For data collection, we define nine cases for the case study, each comprising the coupling of an architectural analysis enabling to specify security characteristics related to *encryption* with a pattern-search-based source code analysis identifying patterns related to weaknesses violating *encryption* and one system to be analyzed. The systems to be analyzed comprise an architectural security design and an implementation including weaknesses that threaten *encryption*. We describe the selection of analyses and the systems in the following. The prototypical implementation of the couplings and all data collected can be found in our data set [30].

1) Analyses Involved in the Case Study: We select architectural analyses specifying security characteristics requiring *encryption* and source code analyses finding patterns that indicate weaknesses that invalidate the *encryption*. We require (meta)models for the input and output of the analyses based on the Eclipse Modeling Framework (EMF) [31], which is used by several model-based analyses such as [5], [10], [16], [26], [32]. We found no pattern-search-based source code analysis providing EMF metamodels. However, the *Parameterization Metamodel* is built with EMF. This is why we can use source code analyses where a transformation of their output into the *Parameterization Metamodel* is possible. We also require the analyses to provide *working tooling*. We define tooling as working if it can either be downloaded and executed without further intervention or we have access to all artifacts and the information on how to build and run it.

a) Architectural Analysis: We select the Access Analysis [33] using input metamodels based on EMF. The Access Analysis extends the Palladio Component Model [34] with *Adversaries*, which *mayKnow* data assigned to *DataSets*. An adversary can gain access *Locations* or *Communication Links*. A vulnerability is reported when an *Adversary* can access an interface in the system or communication link that transmits data the adversary is not allowed to obtain. The security characteristic *encryption* for communication links specifies that communication is designed to be encrypted.

b) Source Code Analyses: We select source code analyses from the OWASP list for static source code analyses [35], a community project for software security, with the following attributes: (1) they find patterns related to weaknesses invalidating encryption in Java source code (2) their results can be exported to machine-readable formats, e.g., JSON and (3) they are free-of-use. This search resulted in the analyses FindSecBugs [13], Semgrep [36] and Snyk [14].

2) Case Study Systems And Injected Weaknesses: We use systems from related work in which encryption requirements are described. We inject weaknesses in the system implementation violating the specified security characteristic indicating requiring *encryption*. We introduce an adversary for all systems that obtains unallowed access to the transmitted data if the specified encryption is violated.

We select JPMail from our example in Section II. The system comprises the components *MailSending*, *POP3*, *SMTP* and *MailReceiving*. We inject the weakness using a weak cryptographic algorithm *DES* as described in the example. The weakly encrypted data is passed over two *encrypted* communication links. Therefore, the coupling should report *four* architectural vulnerabilities. We select the CoCoMe system used by SecDFD [7], representing a shopping system for multiple cash desks. The component *CashDesk* sends the credit card number *cardNumber*, the *storeAccount* and the *amount* of money a customer pays to the component *Bank* via the *requestTransactionService* to handle the payment. This communication has to be *encrypted*. We inject a weakness using the insecure HTTP protocol instead of HTTPS. We expect three vulnerabilities as the weakness will affect all communicated data. Finally, we use the TravelPlanner system of Katkalov et al. [9], representing a flight booking system. The component *TravelPlanner* books a flight by transmitting the *creditcarddata* and the *offerId* of the flight offer *encrypted* to the *Airline* component through the *bookFlight* service. We introduce a weakness by inserting a hard-coded cryptographic key. Only *creditcarddetails* is encrypted with this key, while *offerId* is encrypted with a newly generated key. Thus, only one vulnerability should be reported. In total, we expect *eight* vulnerabilities to be reported by the architectural analysis.

We use SPOON [37] to extract structural information in the implementation for the intermediate structure, such as method calls. This requires the system implementations to use call-return semantics, which does not apply to all systems. We address this by reimplementing these systems as single programs. We generate the structural elements in the implementation according to mappings described by Kramer et al. [38]. We implement the behavior according to behavior descriptions, e.g., sequence diagrams or existing implementation.

TABLE I
EVALUATION RESULTS FOR THE ACCURACY SHOWN BY TRUE POSITIVES t_p, FALSE POSITIVES f_p, FALSE NEGATIVES f_n, PRECISION p, AND RECALL r FOR ALL COUPLINGS. COUPLING CASES ARE REPRESENTED BY (ARCHITECTURAL ANALYSIS, SOURCE CODE ANALYSIS)

Case	t_p	f_p	f_n	p	r
(Access Analysis, FindSecBugs)	$\frac{6}{8}$	1	2	$\frac{6}{7}$	$\frac{6}{8}$
(Access Analysis, Snyk)	$\frac{3}{8}$	1	5	$\frac{3}{4}$	$\frac{3}{8}$
(Access Analysis, Semgrep)	$\frac{6}{8}$	1	2	$\frac{6}{7}$	$\frac{6}{8}$

C. Evaluation Results and Discussion: Accuracy

We applied our coupled analyses with the injected systems on the three case study systems. We first present the results for all cases and interpret them thereafter.

a) Evaluation Results: Table I shows the true positives, false positives, false negatives, precision, and recall for each coupling combined for all systems. The coupling with FindSecBugs, Semgrep and Snyk performs the same modification of the architectural analysis input model in all cases except when applying Snyk to CoCoMe. In contrast to FindSecBugs and Semgrep, Snyk does not report the weakness resulting from using HTTP. Consequently, all results of the coupling are equal except for this case. For the TravelPlanner, every coupling removes the *encrypted* annotation from the communication link. This results in the true positive reported vulnerability for the *creditcarddetails*. However, we also obtain a false positive result for *offerId*, because it is strongly encrypted but affected by the removal of the specified *encrypted* security characteristic due to the weakness of *creditcarddetails*. For JPMail, every coupling reports the vulnerabilities for the first communication link in the *AttackZone*. However, the coupling does not remove the *encrypted* security characteristic for the second communication link. Consequently, no vulnerability is reported while the *header* and *body* passed from *POP3* to *MailReceiving* is still weakly encrypted, which we count as two false negatives. For CoCoMe, the architectural analysis reports vulnerabilities for *amount*, *creditcarddetails* and *account* sent over the communication link after the coupling with FindSecBugs and Semgrep, which we count as three true positives. As Synk does not report the injected weakness, the architectural analysis does not report these vulnerabilities for Snyk. We count the not-reported weaknesses as false negatives, which is conservative as it reduces the recall.

b) Result Interpretation: In the investigated cases, the true positives indicate that the coupling enables the detection of architectural vulnerabilities originating from weaknesses in the implementation. However, the coupling approach negatively affects the precision and recall of the coupled analyses. The decrease in precision, i.e., the false positives, in the TravelPlanner and JPMail system shows the negative influence of removing security characteristics specified for architectural elements that define communication of several services and data, such as communication links. This removal results in interpreting all data communicated as not encrypted rather than only the data affected by the weakness. As a result, the coupling approach works better for security characteristic specified for more fine-granular architectural elements, e.g., direct communication relations like calls or connectors in the architecture. We discuss this limitation in Section VII. This limitation results in *software engineers* to be reported with vulnerabilities that do not exist in the final system and are not reported when performing the architectural analysis in isolation. The false negatives in JPMail show the negative influence of the current search strategies, resulting in architectural vulnerabilities that should be reported but are not. We discuss further research to address this limitation in Section VII. Still, the architectural analysis in isolation would also not detect these vulnerabilities.

We can answer **Q1** positively that the coupling detects architectural vulnerabilities resulting from weaknesses in the implementation, as shown by the true positives. However, we also must answer the **Q2** positively that the coupling introduces *negative effects* on the accuracy (1) by not detecting all architectural vulnerabilities originating from weaknesses in the implementation due to our search strategy and (2) by reporting architectural vulnerabilities that do not originate from the implementation. We subsume that we do not fully achieve **G1: Accuracy**, as in our case study there are variances in precision, ranging between $\frac{3}{4} = 0.75$ and $\frac{6}{7} = 0.85$, and variances in recall, ranging between $\frac{3}{8} = 0.37$ and $\frac{6}{8} = 0.75$.

D. Threats to Validity

The validity of the evaluation is subject to different threats. We structure our discussion by the threat categories *Internal Validity*, *External Validity*, *Construct Validity*, and *Reliability* proposed by Runeson and Höst [39].

Internal Validity ensures that the investigated factors in the evaluation are not influenced by factors the researchers are unaware of. The investigated factors in our evaluation are precision and recall. Tuma et al. [7] discuss the threat of experimenter bias when creating the ground truth. We address this thread by using existing architectural models of systems as far as possible, e.g., the TravelPlanner from Kramer et al. [33] or the architectural model of Seifermann et al. [16] for JPMail.

External Validity concerns how well the findings can be generalized outside the investigated cases. The representativeness of a sample case may be sacrificed to achieve a deeper understanding and better realism of the phenomena under study [39]. Using case studies threatens the generalizability of the results because the transfer to arbitrary other cases cannot be guaranteed. We address external validity using different source code analyses, systems and introduced types of weaknesses. We do not address external validity for architectural analyses, which we discuss as future work in Section VII.

Construct Validity assures that the used metrics can answer the evaluation questions. We can exclude the threat that *precision* and *recall* are unsuitable for measuring the accuracy as they are often used in a wide range of related literature, such as [7], [16], [26], [28], [29].

Reliability ensures that the conducted study, i.e., the data collection and analysis, does not depend on the particular researcher, but other researchers will come to the same results. We address threats by referencing the related analyses and

describe how we designed the systems and collected the metrics. The metrics applied give reasonable evidence and reduce the need for interpretation. In addition, we provide all artifacts of our evaluation, i.e., metamodels, models and the implementation of our coupling approach, in a data set [30].

VII. Limitations and Future Work

Our approach faces limitations from which we derive future work. While our coupling approach is independent of specific security characteristics and weaknesses, we describe and evaluate it for security characteristics relating to *encryption* and weaknesses violating them. We will investigate whether our approach generalizes to other security characteristics and source code analyses. In our evaluation, we use only one architectural analysis. Future work comprises applying our coupling approach to other architectural analyses. The *coupling expert* must select a source code analysis detecting patterns relating to weaknesses violating the security characteristic. Determining this relation between the analyses can be difficult without further guidance, as illustrated by Synk's case and CoCoMe's weakness. This is why we will investigate how to provide an approach for relating architectural analyses and source code analyses for non-compliance analysis. The relation between *Security Characteristics* and *Weaknesses* must be provided manually in our coupling approach. We already applied Large Language Models (LLMs) to retrieve CWEs affecting *encryption* with promising results. In future work, we investigate to what extent LLMs can create relations for the *Security Releations* model. The evaluation shows that the recall of the coupling is influenced by the architectural element annotated with a security characteristic investigated in the coupling. We will research how to limit this influence, e.g., by investigating the propagation of the removal of a security characteristic. Finally, we restrict the search strategy to the first communication relation in the intermediate structure mapping to the architectural design. In future work, we plan to research whether information flow analysis can be used to detect impacts of weaknesses on security characteristics relating to *encryption* throughout the system, similar to SecDFD [7].

VIII. Related Work

We separate related work into the categories *lifting analysis results on another abstraction level* and *analysis coupling*.

Different approaches exist, lifting information in the output of an analysis onto another abstraction level. Related work such as [40], [41] transform design models into a model of an analysis formalism, refactor the model of the formalism based on analysis results and it back into the design model. In contrast, our coupling approach focuses on coupling analyses using only user-facing models [22] available in different development phases, i.e., architectural design and implementation. Similarily, related work such as [42]–[45] analyzes an formalism as an abstraction of the design and liftes the results back to the design space to present the analysis results with design elements known by the analysis user. In contrast, our coupling modifies the architectural analysis input model based on the source code analysis output model rather than only using architectural elements to represent the source code analysis result. Other related work such as [46], [47] perform runtime analysis and update an architectural model to reflect changes for performance or reliability analysis. In the security domain, performing analysis without deploying the system is beneficial to prevent adversaries from exploiting a potential vulnerability before detecting it with a runtime analysis.

The black-box analysis coupling approaches of Dwyer et al. [48] and Cruanes et al. [49] describe how to make results of different tools available without addressing how the analyses interact, i.e., the coupling process. Compared to this, we detail the coupling process in Section IV. The OPAL [50] approach performs grey-box coupling of analyses. This approach is used to combine analysis components [24] on the same abstraction level to form an isolated analysis. In contrast, our black-box coupling approach couples isolated analysis to perform the architectural analysis with details from the implementation.

IX. Conclusion

Architectural security analyses detect vulnerabilities based on an architectural security design model, assuming that an implementation complies with specified security characteristics, like an encrypted communication. If the implementation does not comply with the specified security characteristics, the architectural analysis may miss vulnerabilities in the final system. We present a coupling approach that removes security characteristics specified for architectural elements in the architectural analysis input model based on weaknesses found with a pattern-search-based source code analysis. With this coupling, an architectural analysis calculates vulnerabilities based on information from the implementation rather than on the assumptions made by a software architect. We describe this coupling for security characteristics relating to *encryption* and weaknesses violating *data encryption*. This coupling approach comprises two contributions: **C1** defines a coupling process using a sequential black-box coupling to remove security characteristics specified in the architectural analysis input model based on weaknesses detected by pattern-search-based source code analyses. We face the challenge that the information in the architectural analysis input and the source code analysis output are related but not the same. The source code analysis reports patterns indicating weaknesses while the architectural analysis specifies security characteristics representing a required security-related property. With **C2**, we address this challenge by presenting the usage of intermediate structures that unify information of architectural analyses and pattern-search-based source code analyses.

We perform a case study-based evaluation to investigate the accuracy of the architectural analysis in the coupling regarding weaknesses arising in the implementation. In this evaluation, we create nine cases by coupling of an architectural analysis with three source code analyses and apply them to three systems, each comprising different weaknesses violating the security characteristic *encryption*. We calculate the precision and recall regarding expected vulnerabilities. The evaluation

shows that the architectural analysis can detect architectural vulnerabilities arising from weaknesses in the implementation. However, we also found that our current approach also contains limitations resulting in the coupled analysis to miss vulnerabilities and reports non-existent vulnerabilities. We discuss this limitation and possible future work to address it.

ACKNOWLEDGMENT

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

REFERENCES

[1] S. Hahner *et al.*, "Model-based confidentiality analysis under uncertainty," in *2023 IEEE 20th Int. Conf. on Softw. Archit. Companion*.
[2] H. Shahriar and M. Zulkernine, "Mitigating program security vulnerabilities: Approaches and challenges," *ACM Comput. Surv.*, vol. 44, no. 3, Jun. 2012.
[3] J. Epstein *et al.*, "Software security and soa: danger, will robinson!" *IEEE Security & Privacy*, vol. 4, no. 1, pp. 80–83, 2006.
[4] S. Seifermann, "Architectural data flow analysis for detecting violations of confidentiality requirements," Ph.D. dissertation, Karlsruher Institut für Technologie (KIT), 2022.
[5] K. Tuma *et al.*, "Flaws in flows: Unveiling design flaws via information flow analysis," in *2019 IEEE Int. Conf. on Softw. Archit.*, pp. 191–200.
[6] J. Jürjens, "UMLsec: Extending UML for Secure Systems Development," in *UML 2002 — The Unified Modeling Language*, ser. Lecture Notes in Computer Science. Springer, 2002, pp. 412–425.
[7] K. Tuma *et al.*, "Checking security compliance between models and code," *Softw. and Systems Modeling*, vol. 22, no. 1, pp. 273–296, 2023.
[8] S. Jasser, "Enforcing architectural security decisions," in *2020 IEEE Int. Conf. on Softw. Archit.*, pp. 35–45.
[9] K. Katkalov *et al.*, "Model-driven development of information flow-secure systems with iflow," in *2013 Int. Conf. on Social Computing*.
[10] J. Geismann *et al.*, "Ensuring threat-model assumptions by using static code analyses," in *ECSA 2021 Companion Volume*. CEUR-WS.org.
[11] OWASP top ten | OWASP foundation. [Online]. Available: owasp.org/www-project-top-ten/ (accessed Jan. 04, 2025)
[12] S. Krüger *et al.*, "Cognicrypt: Supporting developers in using cryptography," in *2017 32nd IEEE/ACM Int. Conf. on Automated Softw. Eng. (ASE)*, pp. 931–936.
[13] Find Security Bugs. [Online]. Available: https://find-sec-bugs.github.io/ (accessed Jan. 04, 2025)
[14] Snyk. [Online]. Available: https://snyk.io/de/ (accessed Jan. 04, 2025)
[15] I. Bojanova, "Bug Framework (BF): Formalizing Cybersecurity Weaknesses and Vulnerabilities," National Institute of Standards and Technology, Tech. Rep. NIST Special Publication (SP) 800-231, Jul. 2024. [Online]. Available: https://csrc.nist.gov/pubs/sp/800/231/final
[16] 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.
[17] H. Stachowiak, *Allgemeine Modelltheorie*. Springer, 1973.
[18] T. Stahl and M. Völter, *Model-driven software development : technology, engineering, management*. Wiley, 2006.
[19] M. Brambilla *et al.*, "Model-driven software engineering in practice," *Synthesis Lectures on Soft. Eng.*, vol. 3, no. 1, pp. 1–207, 2017.
[20] C. Talcott *et al.*, *Composing Model-Based Analysis Tools: Foundations*. Cham: Springer International Publishing, 2021, pp. 9–37.
[21] B. Chess and J. West, *Secure programming with static analysis*. Pearson Education, 2007.
[22] C. Talcott *et al.*, *Composing Model-Based Analysis Tools - Composition of Languages, Models, and Analyses*. Springer, 2021, pp. 45–70.
[23] R. Heinrich *et al.*, "A layered reference architecture for metamodels to tailor quality modeling and analysis," *IEEE Trans. Softw. Eng.*, 2019.
[24] S. G. Koch, "A reference structure for modular model-based analyses," Ph.D. dissertation, Karlsruher Institut für Technologie (KIT), 2023.
[25] CWE - Common Weakness Enumeration. [Online]. Available: https://cwe.mitre.org/ (accessed Jan. 04, 2025)
[26] S. M. Peldszus, *The GRaViTY Framework*. Springer, 2022.
[27] V. Basili *et al.*, "The goal question metric approach," in *Encyclopedia of Softw. Eng.*, 2nd ed. John Wiley & Sons, Inc., 2002, pp. 1–10.
[28] M. Walter *et al.*, "Architectural attack propagation analysis for identifying confidentiality issues," in *2022 IEEE 19th Int. Conf. on Softw. Archit.*
[29] 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, p. 259–269.
[30] F. Reiche and R. Heinrich, "Data set of publication on detecting encryption vulnerabilities by coupling architectural analyses and source code analyses," 2025. [Online]. Available: https://doi.org/10.5281/zenodo.14724688
[31] D. Steinberg, *EMF - Eclipse modeling framework /*, 2nd ed., ser. The eclipse series. Addison-Wesley,, 2009.
[32] 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.*, pp. 61–70.
[33] 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
[34] R. H. Reussner *et al.*, *Modeling and Simulating Software Architectures – The Palladio Approach*. MIT Press, 2016.
[35] "OWASP Source Code Analysis Tools List." [Online]. Available: https://owasp.org/www-community/Source_Code_Analysis_Tools (accessed Jan. 04, 2025)
[36] Semgrep. [Online]. Available: semgrep.dev (accessed Jan. 04, 2025)
[37] R. Pawlak *et al.*, "Spoon: A Library for Implementing Analyses and Transformations of Java Source Code," *Software: Practice and Experience*, vol. 46, pp. 1155–1179, 2015.
[38] M. E. Kramer *et al.*, "Change-driven consistency for component code, architectural models, and contracts," in *Proc. of the 18th Int. ACM SIGSOFT Symp. on Component-Based Softw. Eng.* ACM, 2015.
[39] 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.
[40] B. Combemale *et al.*, "A generic tool for tracing executions back to a dsml's operational semantics," in *Modelling Foundations and Applications*. Springer, 2011, pp. 35–51.
[41] V. Cortellessa *et al.*, "From software architecture to analysis models and back: Model-driven refactoring aimed at availability improvement," *Information and Software Technology*, vol. 127, p. 106362, 2020.
[42] 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*, pp. 128–137.
[43] P. Meier *et al.*, "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*, pp. 339–348.
[44] S. Hahner *et al.*, "Modeling data flow constraints for design-time confidentiality analyses," in *2021 IEEE 18th Int. Conf. Softw. Archit. Companion*, pp. 15–21.
[45] C. Gerking *et al.*, "Domain-specific model checking for cyber-physical systems," in *Proc. of the 12th Workshop on Model-Driven Eng. , Verification and Validation (MoDeVVa 2015)*.
[46] R. Heinrich, "Architectural runtime models for integrating runtime observations and component-based models," *Journal of Systems and Software*, vol. 169, p. 110722, 2020.
[47] D. Monschein *et al.*, "Enabling consistency between software artefacts for software adaption and evolution," in *2021 IEEE 18th Int. Conf. on Softw. Archit.*
[48] 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, p. 93–98.
[49] S. Cruanes *et al.*, "Tool integration with the evidential tool bus," in *Verification, Model Checking, and Abstract Interpretation*. Springer, 2013, pp. 275–294.
[50] 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, p. 184–196.