Automatic Derivation of Vulnerability Models for Software Architectures

Yves R. Kirschner*, Maximilian Walter*, Florian Bossert†, Robert Heinrich* and Anne Koziolek*
KASTEL - Institute of Information Security and Dependability, Karlsruhe Institute of Technology
Karlsruhe, Germany
Email: *{yves.kirschner, maximilian.walter, robert.heinrich, anne.koziolek}@kit.edu, †florian.bossert@student.kit.edu

***Abstract*—Software architectures consist of more and more connections to different components or elements. With the increased connection and exchange between different elements also the attack surface increases, since each element might contain vulnerabilities. The vulnerabilities may be harmless on their own, but attackers could develop attack paths from the combination of different vulnerabilities. For a model-based attack propagation analysis, it is useful to have an annotated components model with vulnerabilities. However, depending on the size of the system, the manual annotation of these models is very time-consuming and error-prone. In this context, we present in this paper an approach that automatically annotates vulnerability information to the components of an architectural model. The goal here is to extract security information of source artifacts and transform them into an existing architecture-based security model to enable model-based security risk assessment. We evaluate our approach using three open-source case studies to demonstrate feasibility and accuracy. The results indicate high recall reading vulnerabilities.**

***Index Terms*—Component-based, Context-Aware QoS Model, Modeling and prediction, Software architecture, Security**

I. Introduction

The increase in more and more activities in online areas, such as online banking, also increases the importance of software security [1]. Many security problems are rooted in the design of the software. Therefore, the Open Web Application Security Project (OWASP) introduced in 2021 the new element Insecure Design [2]. It states, that a security issue is based on design flaws or architectural flaws. Identifying these security issues in the software architecture can be helpful because it abstracts from the detailed source code and can consider properties other than the source code such as the deployment. There exist already architectural security analyses, such as [3], [4], which can be used to determine security properties.

However, the creation of software architecture models and security models is very cumbersome and time-consuming. Especially for existing legacy software, there is a high initial modeling effort. While some modeling effort can be reduced by more user-friendly tools, the main effort still exists. Here, automatic recovery approaches might help to reduce the modeling effort.

While classical architecture recovery approaches focus more on extracting the structure, such as components, our architectural attack analysis [5] needs additional information, such as the available vulnerabilities, as input. Software architects can find this information about vulnerabilities in databases, such as the US National Vulnerability Database (NVD) [6]. These databases also provide public interfaces to access the data. Additionally, there exist static source code analyses, which can extract the existing vulnerabilities for components based on the same classifications as the databases. Therefore, recovery approaches could combine this security information with the software architecture and provide automatically created architectural models enhanced with security properties.

Based on this proposed enrichment, our contributions are: *C1)* We propose an approach to extract vulnerabilities with static code analyses from available development artifacts to provide input models for architectural security analyses. *C2)* We propose an approach to link available development artifacts to components of a reverse-engineered software architecture model to enrich these with security information from different data sources.

The resulting architectural model can then be used in security analyses, such as in our attack propagation [5]. This enrichment can help software architects to identify security issues based on the software architecture and might give software architects new insight into existing attack paths.

We evaluated our approach on three open-source case studies (Acme Air, Spring PetClinic, Piggy Metrics) and investigated the properties' feasibility and accuracy regarding the automatic extraction of security vulnerabilities. The results indicate that we can transform vulnerabilities from static code analyses to architecture models. However, some vulnerabilities are overestimated or not all vulnerabilities are transferred.

II. Foundation

Our approach is based on an architectural description language (ADL). We use this ADL for a model-driven attack analysis and combine it with a model-driven reverse engineering approach.

A. Architecture Description Language

As an ADL, we use the Palladio Component Model (PCM) [7]. PCM already provides support for attacker propagation using vulnerabilities [5], [8]. Hence, using it is beneficial since we can reuse the existing metamodel. In addition, PCM supports various quality analyses, such as performance, reliability [7], and other security properties [3]. Supporting components, and provided and required interfaces, PCM aids the component-based development process. The PCM repository defines

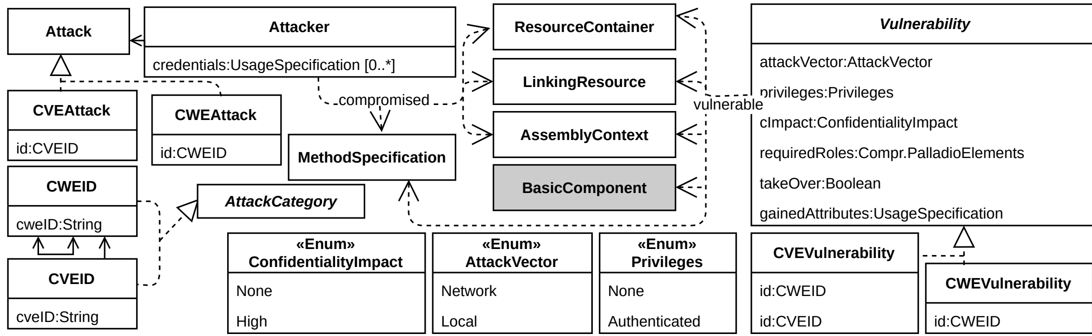

Figure 1: Excerpt of the metamodel for modeling vulnerabilities and attackers based on [5]

components and interfaces. The representation of a component in the PCM is called a `BasicComponent`. `Interfaces` define services, which are implemented by ServiceEffect-Specifications (SEFFs) in `BasicComponents`. The defined components are instantiated in the assembly (or system) model and called `AssemblyContext`. In the resource environment, hardware resource containers representing servers or other processing units and linking resources representing network nodes are modeled. These elements are used in the allocation model to model the deployment of `AssemblyContexts` onto hardware resources.

B. Model-Driven Attack Analysis

In [5], we extended the PCM to support attack propagation based on vulnerabilities and access control. Figure 1 illustrates the extended metamodel for the vulnerabilities and attackers. The excerpt does not contain the access control properties for brevity reasons. The figure is based on UML class diagrams. For simplicity reasons, we left out most of the attributes and show only the most important ones for understanding our approach. We annotate PCM elements (e. g. `AssemblyContext`, `LinkingResource`) with vulnerabilities and defining access control policies for architectural elements such as services or devices. Our vulnerability metamodel is based on commonly used attack classifications. We use the *Common Weakness Enumeration (CWE)* [9] and *Common Vulnerabilities and Exposure (CVE)* [10] to model vulnerabilities (`CWEVulnerability`, `CWEVulnerability`) and attacker capabilities (`CVEAttack`, `CWEAttack`). We further specify the vulnerabilities with properties from the *Common Vulnerability Scoring System (CVSS)* [11] (see fig. 1 `Vulnerability`). We reuse properties such as `Privileges` here, which describe whether an attacker needs to be authenticated or not to exploit the vulnerability. Reusing these properties is beneficial since they are commonly used to classify vulnerabilities and many properties are publicly available in databases such as the US NVD [6]. By using just the values and not the score, we do not rely on the score calculation. In addition, our modeling approach does not necessarily need to use CVSS. Any other approach with similar values can be used. A complete list of the attributes used can be derived from our original publication [5]. For access control, we use attributes called `credentials`. The policies are based on the eXtensible Access Control Markup Language (XACML) [12], which is an industry standard for access control policies. Our analysis then uses this extended PCM to calculate a potential attack graph through the software architecture. This graph exploits access control properties and vulnerabilities. In this work, we want to tackle the manual annotation of vulnerabilities to components. The manual annotation is cumbersome since software architects have to manually identify the vulnerabilities for modeled components. This involves the identification of vulnerabilities on the implementation level. Afterward, they need to trace the implemented source to a modeled component and then manually add the vulnerability properties in the annotation model. For small source code examples, this might be done quickly, however, in bigger systems with multiple components it is harder, and the tracing could be more unclear.

C. Model-Driven Reverse Engineering

The objective of reverse engineering is to identify structures in the form of elements and relations within the software system under investigation. When a reverse engineering approach focuses on recovering models such as architectural models, the task is referred to as Model-Driven Reverse Engineering (MDRE) [13].

In [14], we present a model-driven approach to improve reverse engineering of component architectures. Considering the technologies used, the components with their interfaces are reverse-engineered from software development artifacts such as source code or configuration files. The approach is to model knowledge about the domain of technologies used in component-based software development in order to reverse engineer the architecture from existing text-based artifacts. Text-based artifacts which are considered are written during the development of software systems, e. g., source code or other configuration files such as build configurations.

This modeled domain knowledge captures the impact of a used technology on the architecture of the system. This knowledge might describe how a component is implemented using a particular framework. The approach involves first parsing the existing artifacts and accumulating structure and behavior information about these in models. Then, these

models are analyzed using rules represented by model-to-model transformations, leading to the final transformation of the recognized concepts into a PCM instance.

D. Static Security Code Analysis

Different code analysis techniques can be used to automatically identify vulnerabilities in software products. Some static dependency analyses such as Snyk or similar approaches analyze the dependencies of software and match them with databases for security vulnerabilities. However, they usually cannot find new vulnerabilities, but only identify already known vulnerabilities. So, they can list potentially vulnerable spots in the code, open-source dependencies, or container images. To this end, Snyk provides a command-line interface (CLI) for finding security vulnerabilities. Among other things, the Snyk CLI scans the build configurations of a project, i. e. Gradle[1] `build.gradle` or Maven[2] `pom.xml` files. The CLI can be run locally or in a build pipeline, to check open-source dependencies for security vulnerabilities, for example. However, for smart systems, which often consist of several components, static security code analyze alone cannot provide comprehensive security aspects.

III. Approach

The goal of our approach is to transform architecture-based security models to enable model-based security risk assessment of a component-based system. To this end, we have developed an approach that automatically annotates vulnerability information to the components of an architectural model. The approach supports reverse engineering as soon as code and dependencies are available. Thereby, it detects statically known and publicly available security vulnerabilities. However, the resulting models can also be supplemented later by hand. Figure 2 provides an overview of our approach. Java source code is used as direct input, as well as configurations for build automation using Maven and Gradle. Then, the approach can be divided into two different phases. In the first phase (section III-C), we convert the source code with associated build configurations of a system into its architectural model. In the second phase (section III-D), the information obtained from build configurations is combined with the PCM models to generate the corresponding security models. We first introduce our approach using a motivating example.

A. Motivating Example

The following example briefly illustrates our approach. Listing 1 shows a fragment of the source code for a REST controller implemented using the Spring framework[3]. The Spring framework is an open-source framework for the Java platform that is commonly used for web applications. The `@RestController` annotation indicates that an annotated class is a controller for web requests. The `@GetMapping` annotation indicates the mapping of HTTP GET requests to

[1]https://gradle.org/
[2]https://maven.apache.org/
[3]https://spring.io/

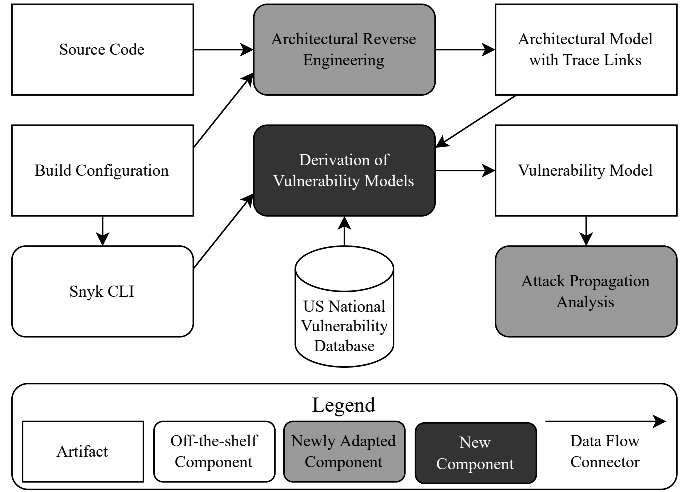

Figure 2: Overview of the key elements of our enhancement architecture for security. From the existing source code and build configurations, our approach automatically extracts a component-based software model and its associated security model. The darkly shaded boxes are the contributions of this paper.

specific handler methods. In a reverse engineering approach [14], that we developed previously and introduced in section II-C, knowledge of these annotations is used to determine the interfaces of a component.

```
@RestController
public class HelloController {
  @GetMapping("/hello")
  public String helloWorld(World world, Model
      model) {
    return "Hello " + world;
  }
}
```

Listing 1: Source code fragment for a REST controller with one handler method implemented using the Spring framework.

The extension of this approach makes it possible to create a trace link between the source code files and the corresponding `BasicComponents` in the PCM. In addition, a similar extension was implemented, which makes it possible to create a link between the source code files and the corresponding build configuration. Listing 2 shows a fragment of the Maven build configuration for the rest controller from listing 1. Using the build configuration, Snyk can now detect a vulnerability based on the dependencies defined there.

Figure 3 shows an example of how this interface implementation can be mapped into the PCM. Snyk returns an ID of the vulnerability. In our case, it is CVE-2022-22965 [15]. Afterward, our analysis looks up the detailed CVSS description on a vulnerability database such as NVD. This description contains for instance the attack vector, in our case this is Network, which means that this vulnerability can be

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
  <version>2.5.4</version>
</dependency>
```

Listing 2: Maven build configuration fragment for a Spring dependency in version 2.5.4.

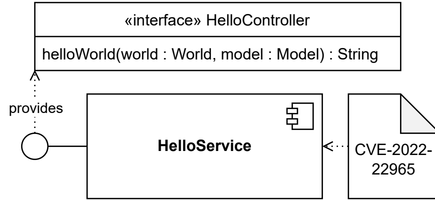

Figure 3: Example mapping of the code from listing 1 into a component with interface definition annotated with vulnerability information based on the dependencies in listing 2.

exploited from any network resource. This information can now be automatically annotated directly to the corresponding `BasicComponents` in the PCM through the link between the build configuration and the source code. Afterward, our attack propagation analysis can use these annotated models to calculate an attack propagation.

B. Extension to the Attack Propagation

Before we can analyze recovered software architectures with our attack propagation, we need to extend our attack propagation analysis. In [5], we only considered instantiated components and did not consider component types in PCM. However, the architecture recovery process can so far only recover component types, called `BasicComponent` in PCM. Therefore, we extended the approach to support vulnerabilities with `BasicComponents`. This extension contains two parts. First, we need to extend the metamodel and second, we need to extend the analysis. For the metamodel extension, we added, similar to the existing vulnerable architectural elements, a link to the repository type (see Figure 1 gray element). For the analysis, we added an initialization step that rolls out the vulnerabilities to the instantiated components. In detail, we first identify for each instantiated component the component type and its modeled vulnerabilities and then create a vulnerability annotation for the instantiated component. This keeps the actual attack propagation algorithm the same as in [5] but considers the vulnerability of component types.

C. Component Model

The first step of our approach is to obtain an architecture model from existing artifacts that provides a structural view of the software system. Artifacts that are written during the development of a software system, i. e. source code or other configuration files, are considered.

For this purpose, we use the approach [14] we developed, which we described in section II-C. Existing artifacts are

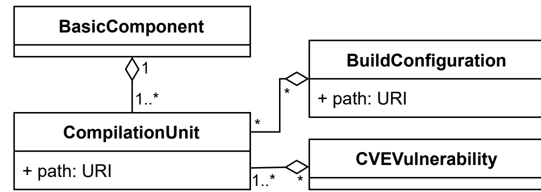

Figure 4: Excerpt of our trace link model with the links of the implementation artifacts to `BasicComponent`, in order to be able to trace back further analysis results from the source code level to the associated architecture level.

analyzed using transformations from text to models. The underlying idea of transformations is to use them to model domain knowledge about technologies. In this context, transformations capture how a particular concept is implemented in a technology and what impact this concept has on the system architecture.

The interfaces of the recognized components identified in this way are then transformed into a PCM instance. This PCM instance is a primarily structural view of the system. For each `BasicComponent`, so-called trace links are stored as references to the original source code. Through this model, any further analysis result can be traced from associated source code or the configuration files to the components at the architecture level.

In order to be able to assign a security vulnerability found in the next step (section III-D) to a detected `BasicComponent`, the existing approach had to be extended by a trace link model. Figure 4 shows the key elements and relationships of the newly introduced trace link model as a UML class diagram.

This model maps the recognized `BasicComponent` in the PCM instance to the underlying compilation units with their unique file path. This means that a `BasicComponent` is always implemented by one to many compilation units. These compilation units can then be assigned any number of build configurations with their unique file path. In a build configuration, the open-source dependencies can be defined for a compilation unit.

D. Security Annotation

The second step of our approach is the annotation of security vulnerabilities to the components of the architecture model. For this purpose, the architecture model is passed with the associated trace links between components and associated artifacts from the first step. The previously generated trace links associate the components recovered from the source code with their associated development artifacts.

These development artifacts, such as source code and configuration files, are then statically analyzed in this step using third-party software. At this point, the CLI of the company Snyk is used. The CLI is executed to find vulnerabilities in the software project. It supports Java with Gradle and Maven among other constructs. The results of this analysis are CVE numbers, each of which is assigned to a development artifact.

For this derived CVE number, further additional required information is then retrieved via the US National Vulnerability Database. If available, in addition to a possible attack vector and privileges, the effects on availability, confidentiality, and integrity are also stored in the security model for the CVE number. This additional information forms the basis for the attack propagation analysis [5].

The artifact that the information is associated with is a build configuration, i. e. a Gradle `build.gradle` or a Maven `pom.xml`, in which abstract information about the project's structure is stored instead of the source code. For the attack propagation analysis, the CVE number, and its associated information should be annotated to the components that they stem from instead of the build configurations that describe those components. To match the vulnerability information to the components, the trace models of the build configurations are used.

E. Limitations

For our approach, we use the Palladio Component Model as a software architecture model. However, the concepts of our approach are also applicable to other software architecture models. Our approach can be adapted easily to use other notations for describing component-based architecture models.

Snyk itself is currently limited to detecting only statically known and publicly available vulnerabilities. Possible unknown attack paths are not detected. At the moment, the vulnerabilities found automatically with Snyk are linked to the architecture based on the own code or dependencies. In future work, other artifacts such as containers and the infrastructure as code will also be considered in the recovery of both the architecture model and the security model.

Snyk compares file signatures against a database of known files to determine vulnerability. If a vulnerability is fixed in code other than by updating dependencies, the vulnerability is still listed. This can lead to an overestimation of practically exploitable vulnerabilities. Similarly, if dependencies are defined for a code file, but are not used in this file, an overestimation also takes place here. However, our approach is designed in such a way that theoretically any analysis tool that links CVEs to code or configuration files can be used.

IV. Evaluation

We structure our evaluation similarly to the Goal Question Metric [16] approach. Our first goal, **G1** is to evaluate the feasibility of our approach. Our evaluation question is **Q1**: *Can we transfer the vulnerabilities from static security analyses to our architectural vulnerability model?* Our metric is a binary with correct and incorrect for each case, whether elements are transferred or not. Our second evaluation goal **G2** is the accuracy of our approach. Accuracy is often used in the evaluation of other architectural approaches, such as [3], [17] to demonstrate the functionality of the approach. Our evaluation question is **Q2**: *How accurately can we annotate found security issues from static code analyses?* Our metrics are precision (p), recall (r) [18], and the harmonic middle $F1$ of both: $p = \frac{t_p}{t_p+f_p}$ $r = \frac{t_p}{t_p+f_n}$ $F1 = 2\frac{p*r}{p+r}$. The t_p are true positives, meaning correctly annotated components. f_p are false positives: incorrectly annotated components, which are not vulnerable but have been reported as vulnerable. Finally, f_n are false negatives: components with undiscovered vulnerabilities. Higher values are better, with the perfect result being 1.00.

Case Study	Java (LOC)	Maven (LOC)	Gradle (LOC)
Piggy Metrics	97 (3292)	10 (684)	-
Spring PetClinic	44 (1180)	8 (828)	-
Acme Air	81 (6207)	1 (56)	2 (75)

Table I: Case studies performed during the evaluation, with the number of relevant files and their sum of code lines.

The third goal **G3** is the applicability of our approach regarding the usage with the attack propagation. The evaluation question **Q3** is: *What manual refinement activities are necessary to use the derived models in the attack propagation analysis?* We try to answer this with a discussion and not with numerical metrics.

A. Evaluation Design

For the evaluation, we used three case studies. Using a case study might increase the insight, show the applicability of the approach, and might increase the comparability [19].

The three open-source case studies conducted during the evaluation are component-based software systems that provide Web services and are built on technologies that are widely used in the industry [20], such as Spring. Table I lists the open-source case studies.

For the feasibility analysis, we performed the following process. One author downloaded each case study from the public code repositories. Afterward, the author performed an automatic architecture recovery with our recovery tool. Then our newly developed annotation tool scans with the help of Snyk for vulnerabilities in components and creates a vulnerability model based on our vulnerability metamodel. This involves the transformation from textual results to models and the extraction from remote databases. Therefore, it is important to syntactically check the generated models. After the automatic steps are finished, two different authors manually evaluated the output models as experts. The two authors were always different persons from the author who used the analyses. During the evaluation process, they investigated the correct syntax of the vulnerability models and whether the constraints of the vulnerability are fulfilled. This covers the creation of matching vulnerability types and the annotation of PCM elements. Therefore, some semantic aspects are also checked. If they both decided that the output was correct, we counted the case as a correct case.

For the accuracy analysis, we reuse the results from the feasibility evaluation. In addition, we used the output of Snyk. It provides the contained vulnerabilities for each build project as a textual output. We compared the annotated vulnerability model to the textual output. We only considered the issues

that have a CVE assigned. If a component in the PCM was annotated and the textual description from Snyk contains the same vulnerability, we count it as t_p. If the textual description does not contain the vulnerability and the vulnerability model has a vulnerability, we count it as f_p. If the textual output contains a vulnerability and the vulnerability model does not, we count it as f_n. We counted this for each case study and then calculated precision, recall and $F1$.

For the applicability, we discuss as the developers of the attack propagation, which additional activities are necessary to use the attack propagation analysis. We focus the activities on creating a valid input model.

Piggy Metrics: Piggy Metrics[4] is intended as an example of the microservice architecture pattern using Spring Boot, Spring Cloud, and Docker. It is designed to demonstrate how these industry-relevant technologies can be used to implement a personal financial advice application. In addition to an authorization service, Piggy Metrics is divided into three core microservices: The account service contains the general input logic and validation, the statistics service performs calculations of key statistics parameters and collects time series for each account, and the notification service stores the user's contact information and notification settings.

We first extracted a PCM from the directly available Java source code and Maven configurations in the public code repository of the Piggy Metrics reference system. This PCM provides a structural view of the system. Based on this, the next step was to fully automatically annotate the components of the architecture model with the security vulnerabilities. For the account service, a total of 146 different CVEs were identified and annotated. The notification service was annotated with a total of 143 CVEs, the authorization service with a total of 146 CVEs, and the statistics service with a total of 146 CVEs.

Spring PetClinic: The Spring PetClinic[5] is an open-source application for exploring and demonstrating technologies and design patterns implemented with the Java Spring Framework. The microservice version of this application consists of the API Gateway, Customers, Vets, and Visits microservices.

First, a PCM was created from the directly available Java source code and Maven configurations in the reference system code repository. In the next step, based on this, the components of the architecture model were fully automatically annotated with the security vulnerabilities. For the API Gateway service, a total of 10 different CVEs were identified and annotated. The Customers service was annotated with a total of 4 CVEs, the Vets service with a total of 4 CVEs, and the Visits service with a total of 4 CVEs.

Acme Air: Acme Air[6] is an open-source benchmark system based on microservices. The system is an implementation of a fictitious airline website. We use the Java-based implementation of the application layer here.

[4]https://github.com/sqshq/PiggyMetrics
[5]https://github.com/spring-petclinic/spring-petclinic-microservices
[6]https://github.com/Acmeair/Acmeair

Case Study	p	r	$F1$
Piggy Metrics	1.00	0.94	0.97
Spring PetClinic	1.00	0.84	0.91
Acme Air	0.39	1.00	0.56

Table II: Evaluation results for accuracy

First, a PCM was created from the directly available Java source code and Maven configurations in the benchmark system code repository. In the next step, based on this, the "morphia" service as a data service implementation was automatically identified and annotated with a total of 2 different CVEs.

B. Result and Discussion

We evaluate the feasibility for each case study. The results are, that for each case study, our approach annotated vulnerabilities to components. However, some vulnerability annotations are duplicated. Nevertheless, this does not affect the later analysis.

For the accuracy analysis, we ignored the duplicated elements as long as the duplicates contain the same information (vulnerability and affected component). Table II shows the result for accuracy. In the first column, the case studies are listed, and then for each case study the precision (p), recall (r), and the $F1$ value. The lowest precision is 0.39 for *Acme Air*, despite that it only has two vulnerability types since the same vulnerability type can affect multiple components. This precision illustrates, that our approach overestimates highly. This overestimation is due to that our approach assigns the found vulnerabilities for the parent project to all the child projects. However, some child projects are not affected. In the future, we have to tackle this issue. The lowest recall is 0.84 for *Spring PetClinic*, because not all CVEs found by Synk were annotated to components. This is due to the fact that when multiple CVEs are specified by Snyk for a vulnerability, our approach only annotated one. In the future, we will need to address this issue as well. Overall, despite the overestimation, the result is promising. In security analyses, identifying security issues can be more important than the additional effort of false positives, since all vulnerable components are identified.

C. Applicability Discussion

While we can automate some parts of the reverse engineering approach, there are some manual refinements steps necessary by the software architect. The manual steps cover four activities: a) creation of the system model and allocation b) specification of access control rights c) refinement of vulnerability properties d) attacker model creation.

For a), software architects have to manually define the instantiated components and their connection. During the automatic reverse engineering of the component and interfaces, all implementations of components contained in the code repository are tried to discover. This covers also, alternative component implementations for test or benchmark purposes, which are contained in a PCM repository. Therefore, a manual

refinement step is necessary, to exclude irrelevant components for the actual running system. In the future, a combination with a dynamic analysis could help. Nevertheless, software architects can reuse the recovered components from our approach. The extended attack propagation (see section III-B) can automatically use the vulnerability from component types.

Regarding b), the attack propagation internally uses a XACML policy decision point and this one can evaluate regular policy files. Therefore, if the runtime system uses XACML policy specifications no manual specification is necessary. If they are not used, software architects or security experts need to manually specify them.

For c), there exist different reasons why a manual refinement could be necessary. One reason could be the manual adaption of the values derived from the CVSS specification. While the vulnerability database contains the classification of a vulnerability in general, this classification might not be correct in every case. For instance, different vendors might compile an open-source product differently and therefore might have different vulnerabilities or properties. Therefore, a software architect might want to adapt the properties. The other part, where a manual refinement is necessary when vulnerability properties cannot be automatically derived from databases. For instance, a vulnerability might leak a certain attribute used in access control policies. These vulnerability properties are system specific and therefore cannot automatically be derived from a universal database.

The last manual refinement step (d)) necessary to run the attack propagation analysis is the creation of an attacker model. The attacker model contains the capabilities of the attacker expressed by CVEs or CWEs they can exploit. Besides the capabilities, the attacker contains also properties about the knowledge or state of the attacker. This contains the initial start point, such as hardware resources, that is similar to the breach point in real-world attacks. While the knowledge and state are system-specific and need to be adapted for each system, the capabilities of the attacker can be reused for similar systems or similar analyses on different systems.

Based on this enriched model, our extended attack propagation analysis can then calculate affected architectural elements from an attack.

D. Threats to Validity

We structured our threats to validity, based on the guidelines for case study research from [21].

Internal Validity discusses whether only the expected factors influence the results. The approach used for determining components with their interfaces returns only one possible view of the system. We have compared these components and interfaces with the existing documentation of the case studies and found discrepancies. However, we cannot say whether these discrepancies are since the documentation is outdated or incomplete in some places.

External Validity discusses how generalizable the results are. Using a case study might increase the insights, but could potentially affect the generalization. To avoid overfitting cases, we used only external case studies. However, the number of case studies is limited, and we only demonstrated the annotation of vulnerabilities. In the future, we want to apply the approach to more external case studies and also evaluate the applicability of the attack propagation analysis.

Construct Validity discusses whether the investigated properties help to answer the evaluation goals. For our approach, this is the relationship between the metrics and goals. The usage of a dedicated evaluation approach like GQM lowers the risk since it shows the relationship between goals and metrics. The first metric is only a binary decision, we assume that it fits well to the feasibility goal. The accuracy goal uses precision, recall, and $F1$, which are often used in related architectural analyses such as [3], [17], [5].

Reliability is about how well other researchers can reproduce the results. For the accuracy result, we use common metrics, which avoids subjective interpretation and increases reproducibility. However, the other goals depend on the experience of the researcher and might be subjective to the researcher. Also, in addition to the possible subjectivity, we publish a dataset containing all our models and raw data [22] and the source code[7] will be publicly available. This helps researchers to reproduce the results.

V. Related Work

We distinguish the related work in approaches related to the attack modeling and analysis, and reverse engineering.

A. Attack Modelling & Analysis

During threat modeling [23], approaches to identify attack paths are often used. Various approach use directed acyclic graphs [24] to model attacks or attackers. One commonly used approach is an attack tree introduced by Schneier [25]. An attack tree defines a goal as the root node and then tasks as child elements to achieve this goal. Another graph-based approach is the CySeMoL [26]. Based on an enterprise architecture, it calculates how secure a system is by using the likelihood of successful attacks. [4], [27] or Deloglos et al. [28] provide an attack path calculation approach. They also use various attack classifications as a basis. Aksu et al. [29] and Yuan et al. [30] provide an attack path calculation by explicitly considering the privilege of the attacker. Kramer et al. [31] developed an attack analysis for the PCM. However, they focus on physical attacks on components. Another model-based approach is UMLSec [32]. It extends UML for security properties and provides various security analyses. These analyses can use a dedicated attacker model. However, they do not consider the propagation of attacks through vulnerabilities. Overall, many approaches use attack modeling and also use reuse existing attack classification. In contrast to them, our attack modeling and analysis approach supports fine-grained access control policies and vulnerabilities in combination with attack propagation. Additionally, we support the automatic

[7]https://github.com/FluidTrust/Palladio-ReverseEngineering-SoMoX-Vulnerability

extraction of known vulnerabilities with our new addition and therefore can partially automate our analysis.

B. Reverse Engineering Approaches

In [33] Raiboulet et al. compare a total of fifteen different model-based reverse engineering approaches. They conclude that these approaches as well as their application domains are diverse. Building on this, MoDisco [34] is the most widely related approach. Brunelière et al. developed with MoDisco a model-based reverse engineering approach that provides support for multiple technologies to generate model-based views of the architecture. Although MoDisco is generic and extensible, it does not support security vulnerability recovery.

VI. Conclusion

In this article, we present a model-driven approach that automatically adds security vulnerability information to the components of a software architecture model. We presented how to link available development artifacts to components of a reverse-engineered software architecture model to enrich them with security information from different data sources. We have also presented how vulnerabilities can be extracted from available development artifacts using static code analysis to provide input models for architectural security analysis. Using our approach is beneficial to reduce the modeling effort in defining and measuring the security of a service in software architectures. We evaluated our approach on three open-source case studies and investigated the properties, feasibility, and accuracy related to the automatic extraction of security vulnerabilities. Feasibility shows that we can transfer vulnerabilities from static code analysis. The accuracy shows that in one case there is an overestimation, but in this case all elements are transmitted and that in two cases not all CVEs associated with a vulnerability are transmitted. In the future, we want to investigate the use of further tools for static code analysis, to be able to look at security vulnerabilities that do not have CVEs. Similarly, we also want to consider other artifacts such as containers and the infrastructure as code when recovering the architecture model and security model in the future. Based on this, we also want to investigate our approach and further case studies, and perform automated security analysis. g

Acknowledgement

This work was supported by funding from the topic Engineering Secure Systems of the Helmholtz Association (HGF), by KASTEL Security Research Lab and the German Research Foundation (DFG) under project number 432576552, HE8596/1-1 (FluidTrust).

References

[1] E. Johns, "Cyber security breaches survey 2021: Statistical releas," *UK Department for Digial, Culture, Media & Sport (DCMS)*, p. 66, 2021.

[2] OWASP. A04:2021 – insecure design. [Online]. Available: https://owasp.org/Top10/A04_2021-Insecure_Design/

[3] S. Seifermann, R. Heinrich, D. Werle, and R. Reussner, "Detecting violations of access control and information flow policies in data flow diagrams," *JSS*, 2021.

[4] N. Polatidis *et al.*, "From product recommendation to cyber-attack prediction: generating attack graphs and predicting future attacks," *Evolving Systems*, vol. 11, no. 3, p. 479–490, Sep 2020.

[5] M. Walter, R. Heinrich, and R. Reussner, "Architectural attack propagation analysis for identifying confidentiality issues," in *ICSA'22*, 2022.

[6] NVD. [Online]. Available: https://nvd.nist.gov/vuln

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

[8] M. Walter, S. Hahner, T. Bures, P. Hnetynka, R. Heinrich, and R. Reussner, "Architectural attack propagation in industry 4.0," *at - Automatisierungstechnik*, 2023, accepted, to appear.

[9] M. Corporation. Cwe. [Online]. Available: https://cwe.mitre.org/

[10] ——. Cve. [Online]. Available: https://cve.mitre.org/

[11] Cvss sig. [Online]. Available: https://www.first.org/cvss/

[12] XACML. [Online]. Available: https://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.html

[13] S. Rugaber and K. Stirewalt, "Model-driven reverse engineering," *IEEE Software*, vol. 21, no. 4, pp. 45–53, July 2004.

[14] Y. R. Kirschner, "Model-driven reverse engineering of technology-induced architecture for quality prediction," in *ECSA-C*, 2021.

[15] NVD. Cve-2022-22965 detail. [Online]. Available: https://nvd.nist.gov/vuln/detail/cve-2022-22965

[16] G. Basili, V. R. Caldiera, and H. D. Rombach, "The goal question metric approach," *Encyclopedia of software engineering*, pp. 528–532, 1994.

[17] R. Heinrich *et al.*, "Architecture-based change impact analysis in cross-disciplinary automated production systems," *JSS 146*, pp. 167 – 185, 2018.

[18] C. Van Rijsbergen and C. Van Rijsbergen, *Information Retrieval*. Butterworths, 1979.

[19] A. van Den Berghe, R. Scandariato, K. Yskout, and W. Joosen, "Design notations for secure software: a systematic literature review," *Softw. Syst. Model.*, vol. 16, no. 3, pp. 809–831, 2017.

[20] F. Gurcan and C. Kose, "Analysis of software engineering industry needs and trends: Implications for education," *International Journal of Engineering Education*, vol. 33, no. 4, pp. 1361–1368, 2017.

[21] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empirical Software Engineering*, vol. 14, no. 2, p. 131, 2008.

[22] Y. R. Kirschner, M. Walter, F. Bossert, R. Heinrich, and A. Koziolek. Dataset: Automatic derivation of vulnerability models for software architectures. [Online]. Available: https://doi.org/10.5281/zenodo.7413806

[23] F. Swiderski and W. Snyder, *Threat Modeling*. Microsoft Press, 2004.

[24] B. Kordy, L. Piètre-Cambacédès, and P. Schweitzer, "Dag-based attack and defense modeling: Don't miss the forest for the attack trees," *Computer Science Review*, vol. 13–14, p. 1–38, Nov 2014.

[25] B. Schneier, "Attack trees," *Dr. Dobb's journal*, vol. 24, no. 12, p. 21–29, 1999.

[26] T. Sommestad, M. Ekstedt, and H. Holm, "The cyber security modeling language: A tool for assessing the vulnerability of enterprise system architectures," *IEEE SYSTEMS JOURNAL*, vol. 7, no. 3, p. 11, 2013.

[27] N. Polatidis, M. Pavlidis, and H. Mouratidis, "Cyber-attack path discovery in a dynamic supply chain maritime risk management system," *Computer Standards & Interfaces*, vol. 56, pp. 74–82, 2018.

[28] C. Deloglos, C. Elks, and A. Tantawy, "An attacker modeling framework for the assessment of cyber-physical systems security," in *Computer Safety, Reliability, and Security*. Springer, 2020, pp. 150–163.

[29] M. U. Aksu, K. Bicakci, M. H. Dilek, A. M. Ozbayoglu, and E. I. Tatli, "Automated generation of attack graphs using nvd," in *ODASPY '18*. ACM, 2018, p. 135–142.

[30] B. Yuan, Z. Pan, F. Shi, and Z. Li, "An attack path generation methods based on graph database," in *2020 IEEE 4th ITNEC*, vol. 1, 2020, pp. 1905–1910.

[31] M. Kramer, M. Hecker, S. Greiner, K. Bao, and K. Yurchenko, "Model-driven specification and analysis of confidentiality in component-based systems," KIT-Department of Informatics, Tech. Rep. 12, 2017.

[32] J. Jürjens, *UMLsec: Extending UML for Secure Systems Development*. Springer Berlin Heidelberg, 2002, vol. 2460, p. 412–425.

[33] C. Raibulet, F. A. Fontana, and M. Zanoni, "Model-driven reverse engineering approaches: A systematic literature review," *IEEE Access*, vol. 5, pp. 14 516–14 542, 2017.

[34] H. Bruneliere, J. Cabot, F. Jouault, and F. Madiot, "Modisco: a generic and extensible framework for model driven reverse engineering," in *Proceedings of the IEEE/ACM international conference on Automated software engineering*, 2010, pp. 173–174.