Architecture-based Attack Path Analysis for Identifying Potential Security Incidents*

Maximilian Walter, Robert Heinrich, and Ralf Reussner

KASTEL – Institute of Information Security and Dependability, Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany
{maximilian.walter,robert.heinrich,ralf.reussner}@kit.edu

Abstract. Analyzing attacks and potential attack paths can help to identify and avoid potential security incidents. Manually estimating an attack path to a targeted software element can be complex since a software system consists of multiple vulnerable elements, such as components, hardware resources, or network elements. In addition, the elements are protected by access control. Software architecture describes the structural elements of the system, which may form elements of the attack path. However, estimating attack paths is complex since different attack paths can lead to a targeted element. Additionally, not all attack paths might be relevant since attack paths can have different properties based on the attacker's capabilities and knowledge. We developed an approach that enables architects to identify relevant attack paths based on the software architecture. We created a metamodel for filtering options and added support for describing attack paths in an architectural description language. Based on this metamodel, we developed an analysis that automatically estimates attack paths using the software architecture. This can help architects to identify relevant attack paths to a targeted component and increase the system's overall security. We evaluated our approach on five different scenarios. Our evaluation goals are to investigate our analysis's accuracy and scalability. The results suggest a high accuracy and good runtime behavior for smaller architectures.

Keywords: Attack Propagation · Software Architecture · Attack Path

1 Introduction

As a society, we digitize various aspects of our lives with new smart devices. This covers different sectors, such as the health sector with a wide variety of eHealth services, the energy sector with smart meters, or production processes

* This work was supported by the German Research Foundation (DFG) under project number 432576552 (FluidTrust), by funding from the topic Engineering Secure Systems of the Helmholtz Association (HGF), by KASTEL Security Research Labs, by "Kerninformatik am KIT (KiKIT)" funded by the Helmholtz Association (HGF), and by the German Federal Ministry of Education and Research (BMBF) grant number 16KISA086 (ANYMOS).

with Industry 4.0. Internet of Things (IoT) devices are the foundation for most of these sectors. These devices exchange data with a wide range of possible services, such as cloud services, thereby building a large and complex network of heterogeneous devices and services.

Often, these devices or services contain vulnerabilities. However, not only IoT devices are affected but also the backend of these systems, such as cloud services or typical company networks with outdated Windows versions [12]. These vulnerabilities can be chained in so-called *advanced persistent threats (APT)* and build complex attack paths and potentially enable attackers to reach critical components, such as payment components [26] or even turn off critical infrastructure, such as the power grid [9].

Analyzing these systems for attack paths is complicated since different devices often have different vulnerabilities. Moreover, these vulnerabilities may manifest in diverse areas of the system, including hardware resources, network resources, and various software components. Therefore, it is essential to model different areas of the system to estimate the potential impact of any vulnerabilities effectively. Software architecture can provide the means to model these different areas. An attack path is then a list of compromised architectural elements. Moreover, a software architecture model may facilitate system analysis, even in cases where a running system is unavailable. Therefore, it enables secure system design and management during development and periods of downtime, such as following an attack or maintenance. Notably, this concept aligns with the principles outlined in the new OWASP Top Ten element "Insecure Design"[18], highlighting that security threats are often embedded within the system design and, therefore, the software architecture. Furthermore, a modeled software architecture enables the creation of what-if scenarios to analyze and find the best solution by modeling and analyzing different scenarios. Existing attack propagation approaches, such as Bloodhound[1], mainly focus on one aspect, such as the Active Directory, or only use a network topology, which often does not contain information regarding software components or deployments. Finally, given the high number of vulnerable elements in many systems, it is not uncommon that there are many possible attack paths that attackers could exploit. In such cases, effective vulnerability management is necessary. One solution is to prioritize and select the most relevant attack paths for mitigation. Therefore, meaningful filter operations are necessary to identify relevant paths. Besides vulnerabilities, attackers may exploit access control policies to gain access to various architectural elements. Once an attacker has gained access to an element, they may use this element to launch further attacks on other elements. Therefore, it is crucial to consider vulnerabilities and access control to develop a comprehensive security analysis for identifying combined attack paths. These attack paths help then to identify potential security incidents, which are multiple unwanted *security events* that threaten the system [11].

In Walter et al. [33], we developed a metamodel and analysis to tackle some of these problems. However, we focused on the propagation of one attacker

[1] `https://bloodhoundenterprise.io/`

from one initial breach point in the software architecture. In contrast, this work focuses on creating multiple attack paths leading to one targeted element in the software architecture. This enables architects to identify potential security risks to critical components. For instance, a software architect could be interested in whether an attack path from an externally accessible component, such as a web service, to a confidential database exists. In addition, this approach estimates the used attacks based on the modeled vulnerabilities and the filtering options. It does not require the concrete modeling of the attacker's capabilities and knowledge as the previous approach [33]. Our contributions to this paper are: **C1**) We extended an architectural vulnerability metamodel by adding support for modeling multiple attack paths leading to a target element and support for filtering options. This enables architects to select attack paths based on the relevant properties, such as the complexity of the used attacks. **C2**) Based on the new extended metamodel, we developed an attack path generation. It generates multiple attack paths to a targeted element and can consider filter options. These filters can help software architects to identify relevant attack paths based on the paths' properties. Additionally, the filters fasten the calculation since they reduce the problem size. In contrast to existing approaches (see Section 5), we consider fine-grained access control policies and vulnerabilities based on the software architecture for attack paths leading to one targeted element. The derived attack paths can help software architects to harden the system.

We evaluated our approach on five scenarios based on real-world breaches and research cases. The investigated properties are accuracy and scalability. The results indicate a high accuracy and acceptable overall runtime for smaller systems. The paper is structured as follows. We describe our metamodel and the attack path generation in Section 2 and Section 3. The evaluation follows in Section 4. Afterward, we discuss related work in Section 5. Finally, Section 6 concludes the paper.

2 Modeling Attack Paths & Path Selection

In Walter et al. [33], we provide a metamodel extension for the Palladio Component Model (PCM) [21] to model access control properties and vulnerabilities. PCM is an architecture description language (ADL) which supports the component-based modeling and analysis for different quality properties, such as confidentiality or performance [21, 25]. We also used the approach to estimate the criticality of the accessed data [34] and analyze different usage and misusage scenarios [32]. The main idea of their approach is to reuse the existing vulnerability classifications Common Weakness Enumeration (CWE) [7], Common Vulnerabilities and Exposure (CVE) [5] and Common Vulnerability Scoring System (CVSS) [6] to describe the vulnerabilities during an attack propagation. These are commonly used to classify vulnerabilities and their attributes can be found in public databases, such as the US National Vulnerability Database (NVD). We also developed an approach to derive the architecture and vulnerabilities automatically [15].

We will explain our approach based on the running example from Walter et al. [33]. Figure 1 illustrates the components, devices, and network entities. The example is settled in an Industry 4.0 setting. It contains a technician who can maintain a machine by accessing a terminal. The machine stores its data on an external storage. This scenario is modeled by three components (`Terminal`, `Machine`, `ProductionDataStorage`). Each of these components is deployed on its own hardware device. A local network connects each hardware device. Additionally, the storage device contains one additional component, which contains confidential data about the production process. For simplicity reasons, we reduced the number of access control policies to two. The `StorageServer` and `TerminalServer` are only accessible by a user with the role `Admin`. Additionally, in our case, we have one vulnerability for the `TerminalServer`. In this scenario, the goal or the target of the attacker is to find potential attack paths, leading to the `ProductStorage` since this component contains confidential data.

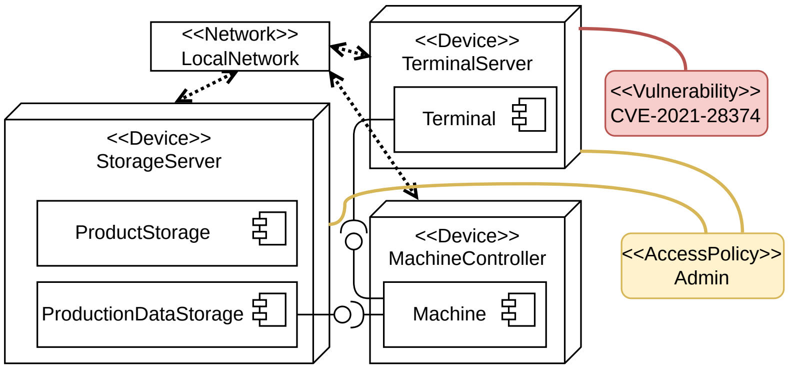

Fig. 1. Running Example with a vulnerable TerminalServer and Access Control policies based on Walter et al. [33]

Figure 2 illustrates the extended vulnerability metamodel. The gray elements are the original metamodel elements [33], the black ones are PCM elements, and the white elements are the new elements. For simplicity reasons, we left out non-relevant elements for this approach. The complete metamodel can be found in our dataset [35]. The main element to integrate vulnerabilities in PCM is the `Vulnerability` class. It annotates `LinkingResources` for network resources, `ResourceContainers` for hardware devices and `AssemblyContexts` for instantiated components with vulnerability information. This element is implemented by two concrete elements the `CWEVulnerability` and the `CVEVulnerability`. The `CWEVulnerability` describes more general vulnerabilities based on a CWE class, and the `CVEVulnerability` describes a concrete vulnerability. The relationship between CVE and CWE is not represented in the model excerpt but is included in the metamodel. The `Vulnerability` has attributes, such as the attack vector (the location from which a vulnerability is exploitable) or the gained attributes through the exploitation.

While our previous metamodel can already model vulnerabilities and access control properties, the output is restricted to one attack propagation graph for a list of concrete attacks. It cannot represent different attack paths leading to a target or attacks limited by their properties. Hence, we need to add support for different attack paths. Additionally, for identifying the relevant attack paths, we need to add support for finding the relevant attack paths.

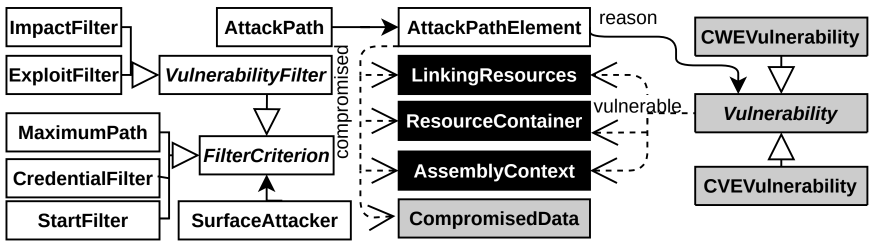

Fig. 2. Excerpt of the extended metamodel with filters (white elements are new elements, gray ones are taken from Walter et al. [33], and black ones are taken from PCM)

The starting point for an attack path is the attack's initial start point, and the targeted element is the element an attack wants to infiltrate. In our running example (Figure 1), the targeted element is the `ProductStorage`. It is, therefore, the element to which all the attack paths should lead. The start point of the list is connected by its elements to the targeted element. The connection is realized by vulnerabilities or exploited credentials. The attack path is represented with the `AttackPath` element. The actual path elements are modeled as a list of `AttackPathElements`. Each `AttackPathElement` describes a compromised architectural element and stores the reason for the compromisation.

With the extension to the metamodel so far, our analysis can calculate attack paths leading to one targeted element based on the modeled software architecture. However, even in our small example, this could lead to many irrelevant attack paths. For instance, for our running example, we get an attack path to the targeted element for every architectural element, resulting in seven paths. In larger systems, this might be even more. Many attack paths may be irrelevant because they demand initial knowledge about specific credentials. For instance, in our running example, there is an attack path from the `MachineController` to the `ProductStorage` over the `StorageServer`. However, this path would require the knowledge of the `Admin` credentials. Usually, a system architect might assume that the admin's credentials might be secure. Therefore, the attack path can sometimes be considered irrelevant.

The selection of relevant paths is realized by filtering. We currently support five filter options. The common abstract element to model attack path filters is the `FilterCriterion`. The filters are then realized as child elements, allowing an easy metamodel extension for new filters. The first filter is the `MaximumPath` filter, and it restricts the path length of the found attack paths. This property can also be found in related approaches, such as [20]. This is beneficial if software architects are only interested in short attack paths because they may be simpler than longer paths. As described, in some cases, it is beneficial to restrict the initial usage of credentials. This is represented by the `CredentialFilter`. Suppose the software architect is only interested in an attack path from certain elements, such as in our running the externally accessible `Terminal`, to the target element. They can use the `StartElement` filter in that case.

The last two filters (`ImpactFilter`, `ExploitFilter`) use properties of the vulnerability for filtering. Hence, they are grouped together with the common parent `VulnerabilityFilter` element. Because the initial metamodel does not include all CVSS properties, we added the following: 1. *AttackComplexity* describes how complex it is to exploit the vulnerability. 2. *UserInteraction* describes that the attacker needs additional support from the user to exploit the vulnerability. 3. *IntegrityImpact* is the impact regarding integrity. 4. *AvailablityImpact* is the impact regarding availability. A more detailed description of the properties can be found in [6]. The `ExploitFilter` filters attacks based on the attackVector, attackComplexity, UserInteraction. This enables an architect to find only attack paths to a targeted element, which contains easily exploited vulnerabilities. This can be helpful in considering different attacker types. The `ImpactFilter` filters out vulnerabilities of a certain impact, such as only attack paths that affect the confidentiality of a system.

The different filters are then selected in the `SurfaceAttacker`. It stores a list of the filter criteria. Additionally, it contains the information necessary to calculate an attack path by storing the targeted element.

3 Attack Path Identification

Based on a modeled software architecture, our approach can identify attack paths leading to a targeted element. We identify the potential attack paths based on an attack graph. In contrast to our previous work[33], the attack graph contains all the vulnerabilities as long as they do not share the filtered properties. The graph is especially not limited by a set of specified attacks. Figure 3 illustrates an attack graph based on our running example. The graph consists of vertices, which are the vulnerable architectural elements. In our case, these are elements from the type `LinkingResource`, `ResourceContainer`, and `AssemblyContext`. The edges are possibilities to compromise a vertex from another vertex. For this, the original architectural elements represented by the vertex need to be connected. For instance, this could be the network connection like in our running example with the `MachineController` to the `Terminal-Server`. Additionally, the edges have three types. The first type models the necessary credentials to access a vertex. For instance, in our running example, the `TerminalServer` is connected with the `MachineController` and the `Admin`

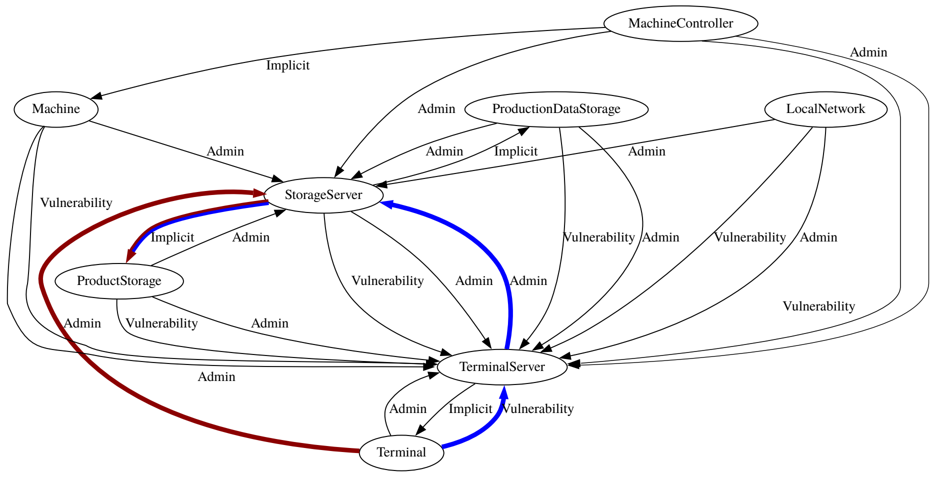

Fig. 3. Attack Graph of the running example with attack path p1 in red and attack path p2 in blue

property gives access to the `TerminalServer`. Therefore, there is a directed edge from the `MachineController` to the `TerminalServer` with the label `Admin`. The second edge type models vulnerabilities, which can be exploited on the target vertex. For instance, the `Terminal` is deployed on the `TerminalServer` and the `TerminalServer` is vulnerable to `CVE-2021-28374`. Therefore, a vulnerability edge exists between `Terminal` and `TerminalServer`. In our illustration, we renamed `CVE-2021-28374` with vulnerability. However, in the analysis, the edge still has the information about its vulnerability. The last edge type is implicit edges. These are edges between a `ResourceContainer` and the components deployed on it. Here, we assume that a compromised hardware automatically compromises the underlying software. For instance, in our running example, the `ProductStorage` is deployed on the `StorageServer`. Hence, there is a directed implicit edge from `StorageServer` to the `ProductStorage`.

Algorithm 1 illustrates the graph creation process in more detail. For the graph creation, we need the software architecture (`arch`) and the selected filters (`fil`). We first create an empty graph (l. 2) and then iterate over all relevant architectural elements (`ResourceContainer`, `AssemblyContext`, `LinkingResource`). For each element, we identify the connected neighbors (l. 4). Afterwards, we iterate over the neighbors and identify how they are connected. If the neighbor element is deployed on the current element, we add an implicit edge (l. 6/7). If the neighbor has access control policies, we add an edge containing this policy. The last step is to identify whether the neighbor has a vulnerability (l. 12). If it has at least one vulnerability, we iterate over all the vulnerabilities of the neighbor. We check whether each vulnerability's attack vector is within the connection vector to the current architectural element (l. 14). Besides the attack vector,

Algorithm 1 Simplified Attack Graph Creation

```
 1: procedure ATTACKGRAPHCREATION(arch,fil)
 2:     $g := emptyGraph()$
 3:     for all $res := arch.getElements$ do
 4:         $neighbours := getConnectedElements(res)$
 5:         for all $n := neighbours$ do
 6:             if $isDeployment(res, n)$ then
 7:                 $g.createImplicitEdge(res, n)$
 8:             end if
 9:             if $n.hasACPolicy()$ then
10:                 $g.createACEdge(res, n)$
11:             end if
12:             if $n.hasVulnerability()$ then
13:                 for all $vul = n.getVulnerability()$ do
14:                     if $isInAVector(vul.AVector())$ &&
15:                      $fil.notFiltered(vul)$ then
16:                         $g.createVEdge(res, n, vul)$
17:                     end if
18:                 end for
19:             end if
20:         end for
21:     end for
22: end procedure
```

we also check whether the vulnerability can be filtered based on the use of `VulnerabilityFilters`. For instance, a `ExploitFilter` with a selected high complexity would create an attack graph that contains only vulnerabilities with low attack complexity. This is helpful in scenarios where software architects want only to consider low-complexity attacks.

Based on this attack graph, the attack paths are calculated by calculating the path from a node to the targeted node. An attack path is a sequence of nodes that are connected by edges. It has a starting node and a target node from the targeted architectural element. For instance, based on the attack graph in Figure 3, an attack path (p1) with the targeted element `ProductStorage` could be: `Terminal` $\xrightarrow{\text{Admin}}$ `StorageServer` $\xrightarrow{\text{Implicit}}$ `ProductStorage` The attack path is also highlighted in red in Figure 3. Besides the start point, the path on the graph and the endpoint, an `AttackPath` also contains a set of initially required credentials. These can be calculated by first getting all required credentials. The required credentials are all credentials that are on the edge of an attack path. In our running example, this is the `Admin`. Afterward, all credentials gained during the attack path are removed. The rest are then the initially required credentials. Since we do not gain any credentials in our example path, the `Admin` attribute is in the initial required set.

The actual attack path is calculated by first determining the start nodes. The start nodes are all nodes except the target node. If a `StartFilter` exists, only these elements are start nodes. Afterward, the analysis calculates paths to the target node. The path finding invalidates solutions, which require the filtered credentials as initial credentials. However, it supports the gaining of credentials during the path. Therefore, the path can contain the filtered credential. After a path is found, we check for the length of the path. We discard it if it exceeds the length specified in the `MaximumPath` filter. Otherwise, we add the path to the list of attack paths. For our running example with the `ProductStorage` as a target, we get seven attack paths, including our previous example, p1. However, not all are reasonable. For instance, the attack path from the `StorageServer` only exploits the deployment relationship. A software architect can specify a start filter to get a better solution. We choose a `StartFilter` containing only the `Terminal` since external technicians can access it. Then the output is only the attack path p1. However, this attack path requires that the attacker knows the `Admin` credentials since it is an initially required credential. While an attacker could have it at the beginning, in general, we assume that an attacker does not have the knowledge. Therefore, we create a `CredentialFilter` with the `Admin`. If we run our analysis now, we get the following attack path (p2): `Terminal` $\xrightarrow{Vulnerability}$ `TerminalServer` $\xrightarrow{Admin}$ `StorageElement` $\xrightarrow{Implicit}$ `ProductStorage`. The path is highlighted in blue. This attack path still uses the `Admin` credential but gains it by exploiting the vulnerability and, therefore, does not require it from the start. A software architect can then use the resulting attack path and consider mitigating the attack path or accepting the risk.

4 Evaluation

We structure our evaluation using the Goal Question Metric [3] approach. Afterward, we will explain our evaluation scenarios, design and discuss our results, threats to validity, and limitations.

Goals, Questions, Metrics The first evaluation goal **G1** is to investigate accuracy. Accuracy is an important property that is also investigated in other related approaches, such as [25, 10]. The evaluation question **Q1** is: *How accurately does the analysis identify the attack paths?* This question is important since a low accuracy suggests that our analysis does not work adequately and that the attack paths might be meaningless for software architects. Our metrics are precision (p), recall (r) [31] 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 detected attack paths. f_p are false positives that are attack paths, which are actually no attack paths and f_n false negatives are not found attack paths. Higher values are better.

Our second goal **G2** is to evaluate the scalability of the approach. The number of architectural elements is increasing due to trends like IoT. Furthermore, new vulnerabilities are discovered continuously during the system's lifespan. Hence, continuously searching for the system's existing attack paths is necessary. One possible solution is to conduct checks similar to integration tests as recommended by [24]. Typically, these tests run daily. Therefore, it is required that the analysis is completed within a few hours. Our questions are: **Q2.1** *How does the runtime of the graph creation behave with an increasing number of elements?* **Q2.2** *How does the runtime of the path finding behave with an increasing number of elements?* We split the evaluation into two questions to investigate the goal in more detail. Q2.1 investigates the part where the analysis transforms the software architecture in an attack graph, and Q2.2 then covers identifying an attack path on a given attack graph. The G2's metric is the relation between runtime and input elements.

Evaluation Scenarios We answer our evaluation question based on five scenarios. Two scenarios (Target, Power Grid) are based on real-world system breaches. One scenario is based on the research case TravelPlanner [14] and one scenario is based on the cloud case study from [2]. The last scenario is based on our running example. The Target, Power Grid and TravelPlanner scenarios are also used in Walter et al. [33]. Hence, the architectural models are the same. Using these scenarios for our evaluation increases the insight and illustrates the applicability and comparability of our approach. Table 1 illustrates some characteristics of the evaluation scenarios. It contains the name of the scenario, the number of instantiated components (abbrev. comp), the number of hardware resources (abbrev. hard.), the number of linking resources (network) and the number of potential attack paths. In addition, it contains the evaluation results for G1.

Table 1. Characteristic of the evaluation scenarios and results for accuracy goal

Scenario	**comp.**	**hard.**	**network**	**paths**	p	r	$F1$
Target	7	6	2	14	1.00	1.00	1.0
Power Grid	9	8	2	16	1.00	0.88	0.93
Cloud Storage	11	16	4	14	1.00	1.00	1.00
TravelPlaner	4	3	3	1	1.00	1.00	1.00
Maintenance	4	3	1	7	1.00	0.86	0.92

Our first scenario is a scenario based on the Target breach, which involved attackers stealing access credentials from a supplier to access Targets's billing business backend. Afterward, they exploited different vulnerabilities in other components to gain access to unencrypted credit card data. The model is based on [26, 19]. It contains POS devices, FTP storage servers, and databases annotated with vulnerabilities, as they were components compromised in the Target breach [26]. These elements are segregated by the supplier in a separate network. The targeted element is one POS device. The second scenario is based on the cyberattack on the Ukrainian Powergrid in 2015, which resulted in a widespread power outage. The model relies on the report of [9] and covers the attack propagation from the back-office network to the ICS network. The target is the circuit breakers in the ICS network. The third scenario is a research case study for threat modeling in a cloud environment [2]. In contrast to the previous scenarios, it is not a real-world system breach. However, the software architecture is based on concepts and ideas from real-world products. It resembles a cloud storage environment. We manually created a PCM model based on the description for the first proposed cloud infrastructure. Here, the targeted element is a database. The fourth scenario is the confidentiality research case study TravelPlanner [14]. It is used to evaluate different security analyses, such as [25, 33]. The previous scenarios are based on real-world breaches or inspired by real-world cloud centers. Therefore, the attacker behavior was given by the case. This case study lets us define the attacker's behavior in more detail. The case is a simple mobile application to book flights. It has four entities: customer, credit card center, travel agency, and airline. The fifth scenario is our running example, the maintenance scenario.

Evaluation Design For the accuracy analysis, we used the five scenarios. We manually determined the number of attack paths for each scenario based on their descriptions. We tried to find an attack path from each architectural element (excluding the targeted element) to the targeted element. For the attack paths, we used the vulnerabilities and potentially found credentials. However, we excluded as initial credentials all credentials so that an attack path needs to either find the necessary credentials or need to exploit a vulnerability. This is beneficial to get more complicated attack paths than otherwise the attack path could be just using the root or admin credentials. For each scenario, we then manually checked whether each attack path was a valid attack path. An attack path is valid if it is a list of connected vulnerable elements from the start point to the target and each vulnerability can be exploited by the attacker. If it is a valid path, we count it as t_p. If it is not a valid path, we count it as f_p. If we found for one architectural element an attack path and the analysis did not show an attack path, we counted this as f_n. Based on these values, we then calculate p, r and $F1$ for each scenario.

For analyzing the scalability, we first identified the influencing factors. We separate this along Q2.1 and Q2.2. For Q2.1, based on our algorithm 1, the most influencing factors are l. 3, 5, and 13 because of their loops. The other lines, 2, 4, 6-12,14-17, are not relevant. For l. 3,5, the relevant attributes are the architectural elements (l. 3 and l. 5). For l. 13, it is the vulnerabilities for an architectural element. For the former, we choose to scale along the number of connected resource containers. This creates a linked chain of vulnerable resource containers and creates a worst-case scenario for the graph creation. The behavior for architectural elements is similar in the algorithm, so the analysis time should be similar. We did not investigate the scaling along the number of vulnerabilities for one architectural element because, usually, the number for one element is not very high. We measured the runtime starting from the graph creation with the already loaded PCM models till the attack graph is returned. We scaled by the power of 10 from 10^1 elements to 10^5 elements. Regarding Q2.2, the relevant factors are the number of edges between distinct nodes and the path length. We achieve the first by choosing the scaling along the `ResourceContainer`. For the second, we use a start filter and set it to the first element in the chained elements, and the target element is the last. This will force a worst-case scenario for a single path. We assume that software architects are more interested in only a filtered list for bigger architectures. For instance, similar to our running example with the `Terminal`, they are only interested in paths from externally accessible components to certain internal components. Here, we measured the time after the attack graph from Q2.1 is created till one attack path is returned. For both, we repeated each measurement five times and calculated the average to avoid outliers. We performed one warm-up analysis and run the analysis on a Debian 11 VM with 21 AMD Opteron Processor 8435 with 62.5 GB RAM.

Results & Discussion Accuracy The last three columns in Table 1 show the evaluation results for G1. For each scenario, we get the perfect precision of 1.00. This means that every attack path of the analysis was an actual attack path regarding our manual comparison. We archived these perfect results since the cases are small, and we focused on a restricted model with no dependencies to unknown behavior, which simplifies the results. Regarding the recall, we archive in the Cloud Storage, Target, and TravelPlanner scenarios the perfect results of 1.00 and also an F1 score of 1.00. This means that our analysis can find all the attack paths from our manual comparison in these scenarios, and they are valid attack paths. However, in the Power Grid, our analysis missed two attack paths and in the Maintenance scenario, one attack path. Therefore, we only have a recall of 0.88 and 0.86 in these scenarios. The F1 score is 0.93 and 0.92. The missed attack paths can be traced back to our usage of simple paths during the attack path creation. For simplicity and performance reasons, the attack paths are loopless and do not contain duplicates. However, in the missed cases, it would be necessary to have loops to get the required credentials. For instance, in the maintenance scenario, the attack path from the `TerminalServer` would require one self-loop to get the necessary credentials.

Results & Discussion Scalability Figure 4 illustrates the scalability results. The horizontal axis shows the number of resource containers and the vertical axis shows the runtime in ms. The blue line with circles is the graph creation and the red line with the boxes is the path finding. Both axes use a logarithmic scale. The runtime of both functions is very close together. For 10 elements, the graph creation needs around 26ms and the path finding around 42ms. It then slowly increases till around 10^3 elements with 597ms (graph creation) and 693ms (path finding). From there, the runtime grows longer until it takes around 5.7×10^6ms (graph creation) and 5.6×10^6ms (path finding) for 10^5 elements. This summarizes to a runtime of around 3 hours. The scalability behavior is not ideal. However, the runtime should still be sufficient for the usage in daily analysis runs. In addition, the model sizes with 10^5 elements are quite high. Usually, the model sizes are smaller. Even in bigger architectures like in IoT environments, the architecture can be reduced by grouping similar elements, for instance, when there are groups of sensors connected to the same backend.

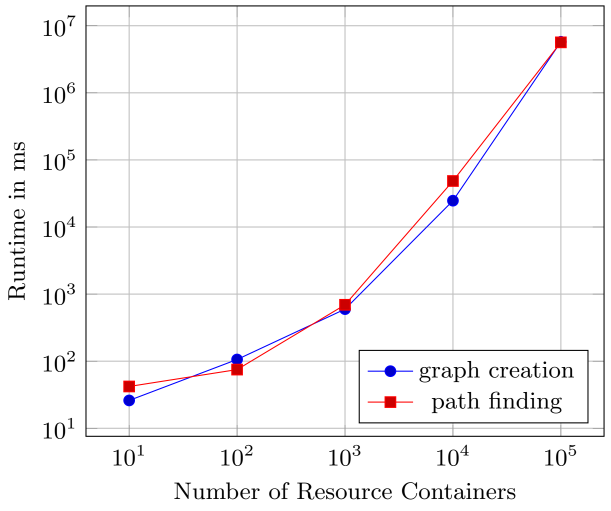

Fig. 4. Scalability results for increasing number of resource containers

Threats to Validity We structure our threats to validity on the guidelines for case study validity from Runeson and Höst [22].

Internal Validity is that only the expected factors influence the results. Our evaluation depends on the modeled system as input and the results reflecting modeled properties. Especially since we also manually created the reference output. We tried to lower the risk by using real-world breaches and literature to create the reference output. For the scalability, other factors, such as the general system usage, could affect the runtime. To avoid this, we repeated the experiment 5 times and ran it on a separate VM. We used multiple real-world breach and research scenarios in our evaluation to ensure the *External Validity* of our results. While we modeled systems are small, we covered all relevant model elements for our extension and analysis features. While scalability may vary with different architectural elements, our internal handling of elements should produce similar results. *Construct Validity* is about the validity of the investigated properties for the intended goal. In our case, the properties are the metrics, and the goal is the evaluation goals. To lower the threat, we used the GQM approach, which illustrates the connection between goals and metrics. For accuracy, we use precision, recall and F1. These metrics are often used to describe the accuracy in different related architectural approaches such as [25, 33]. Therefore, we assume the metrics to be appropriate and the risk to be low. The scalability metric is a simple runtime metric and similar metrics are used in related approaches, such as [20]. *Reliability* discusses how reproducible the results are by other researchers. We use metrics to answer our evaluation question, avoiding subjective interpretation and increasing reproducibility. Besides the metrics, we also provide a dataset [35] for others to verify the results.

Limitations Our approach requires an architecture model and the manual creation of the vulnerability model. Our approach can identify attack paths only based on known vulnerabilities. In addition, it can only be used to identify mitigation locations, but does not support advanced mitigations, such as trusted execution environments. While we already consider the involvement of third parties in our filter, the actual attack calculation does not consider it besides in the filtering.

5 Related Work

We divided the related work in the section of *Policy Analysis*, *Model-based Confidentiality Analysis* and *Attacker Modeling.*

Policy Analysis Our approach analyzes access control policies to estimate the necessary credentials for an attacker. Other approaches can consider various other policy quality aspects. One policy analysis approach is Margrave [8], which can calculate change impact on policies. Another policy analysis is Turkmen et al. [30], which uses SMT internally to analyze policies for different properties, such as change impact and attribute hiding. In summary, all the approaches mentioned focus on policy analysis, not attack propagation.

Model-driven Security Analyses Our approach uses model-driven concepts for generating the attack paths. UMLSec [13] extends UML with security properties. It adds different analysis types, such as secure communication link, fair exchange, and confidentiality. Additionally, they include an attacker model for checking the security requirements. In contrast, our approach focuses on the attack path generation. Another UML extension for security is SecureUML [17]. They focussed on access control. So far, they do not support attack propagation or attack path generation. There exist various approaches which analyze information flow or access control based on some model, such as SecDFD [29], and Data-centric Palladio [25]. In contrast, both use dataflow definitions, but do not consider attack paths calculation. Attacker-related approaches are the Sparta approach [27], Berger et al. [4] or Cyber Security Modeling Language (CySeMoL) [28]. The first two are dataflow analyses in threat detection. In contrast to our approach, these focus on single threat detection and not combining different vulnerabilities/threats to attack paths. CySeMoL [28] calculates potential attack paths but does not use a fine-grained access control system.

Attacker Modeling Schneier [23] introduced the idea of attack trees, which are used in many approaches to model attacker behavior [16]. Polatidis et al. [20] present an approach for attack path generation. Other approaches are, for instance, Aksu et al. [1] and Yuan et al. [36]. In contrast to our approach, all the mentioned approaches use a network layer perspective instead of a component-based software architecture and do not consider fine-grained access control policies.

6 Conclusion

We proposed an approach for generating potential attack paths to a targeted architectural element. Our presented metamodel extension enables architects to model filtering options for attack paths and specify targeted elements. In contrast to Walter et al. [33], our attack analysis provides multiple attack paths to the targeted elements and can remove non-relevant paths by using the filter options. The evaluation indicates that our approach can find in several scenarios attack paths with high accuracy and for smaller systems within a reasonable time. Our approach can help to identify potential weak spots in the software architecture. Software architects can use this information to add mitigation mechanisms to harden the system and prevent attacker propagation. In the future, we want to investigate the problem with the missing attack paths in the evaluation. Additionally, we want to consider mitigation approaches and combine the approach with dataflow analyses similar to [34].

Acknowledgement

We like to thank Jonathan Schenkenberger, who helped to implement this approach during his Master's thesis.

References

1. Aksu, M.U., *et al.*: Automated Generation of Attack Graphs Using NVD. In. CODASPY, pp. 135–142. ACM (2018)
2. Alhebaishi, N., *et al.*: Threat modeling for cloud data center infrastructures. In: Foundations and Practice of Security, pp. 302–319 (2016)
3. Basili, G., *et al.*: The goal question metric approach. Encyclopedia of software engineering (1994)
4. Berger, B.J., *et al.*: Automatically extracting threats from extended data flow diagrams. In: ESSoS, pp. 56–71 (2016)
5. CVE, `https://cve.mitre.org/` (visited on 11/01/2022)
6. CVSS SIG, `https://www.first.org/cvss/` (visited on 11/01/2022)
7. CWE, `https://cwe.mitre.org/` (visited on 11/01/2022)
8. Fisler, K., *et al.*: Verification and change-impact analysis of access-control policies. In: International Conference on Software Engineering 2005, p. 196 (2005)
9. Hamilton, B.A.: Industrial Cybersecurity Threat Briefing. Tech. rep., p. 82
10. Heinrich, R., *et al.*: Architecture-based change impact analysis in cross-disciplinary automated production systems. JSS 146, 167–185 (2018)
11. ISO: Information technology. en. Standard ISO/IEC 27000:2018, Geneva, CH (2018)
12. Johns, E.: Cyber Security Breaches Survey 2021: Statistical Release. (2021)
13. Jürjens, J.: UMLsec: Extending UML for secure systems development. In: UML, pp. 412–425 (2002)
14. Katkalov, K.: Ein modellgetriebener Ansatz zur Entwicklung informationsflusssicherer Systeme. doctoral thesis, Universität Augsburg (2017).
15. Kirschner, Y.R., *et al.*: Automatic Derivation of Vulnerability Models for Software Architectures. In: IEEE ICSA-C, pp. 276–283 (2023)

16. Kordy, B., *et al.*: DAG-based attack and defense modeling: Don't miss the forest for the attack trees. Computer Science Review 13–14, 1–38 (2014)
17. Lodderstedt, T., *et al.*: SecureUML: A UML-based modeling language for model-driven security. In: UML 2002, pp. 426–441 (2002)
18. OWASP Top Ten Web Application Security Risks — OWASP, `https://owasp.org/www-project-top-ten/` (visited on 11/01/2022)
19. Plachkinova, M., and Maurer, C.: Security Breach at Target. Journal of Information Systems Education 29(1), 11–20 (2018)
20. Polatidis, N., *et al.*: From product recommendation to cyber-attack prediction: generating attack graphs and predicting future attacks. Evolving Systems 11(3), 479–490 (2020)
21. Reussner, R., *et al.*: Modeling and Simulating Software Architectures – The Palladio Approach. MIT Press (2016). ISBN: 9780262034760
22. Runeson, P., and Höst, M.: Guidelines for conducting and reporting case study research in software engineering. Empirical Software Engineering 14(2), 131 (2008)
23. Schneier, B.: Attack trees. Dr. Dobb's journal 24(12), 21–29 (1999)
24. Securing the Software Supply Chain: Recommended Practices Guide for Developers, p. 64. Cybersecurity and Infrastructure Security Agency (CISA) (2022)
25. Seifermann, S., *et al.*: Detecting Violations of Access Control and Information Flow Policies in Data Flow Diagrams. Journal of Systems and Software 184 (2021)
26. Shu, X., *et al.*: Breaking the Target: An Analysis of Target Data Breach and Lessons Learned. arXiv:1701.04940 [cs] (2017)
27. Sion, L., *et al.*: Solution-Aware Data Flow Diagrams for Security Threat Modeling. In: Symposium on Applied Computing, pp. 1425–1432. ACM (2018)
28. Sommestad, T., *et al.*: The cyber security modeling language: A tool for assessing the vulnerability of enterprise system architectures. IEEE Systems Journal 7(3), 363–373 (2012)
29. Tuma, K., *et al.*: Flaws in Flows: Unveiling Design Flaws via Information Flow Analysis. In: International Conference on Software Architecture 2019, pp. 191–200
30. Turkmen, F., *et al.*: Analysis of XACML Policies with SMT. In: Principles of Security and Trust, pp. 115–134. Springer (2015)
31. Van Rijsbergen, C., and Van Rijsbergen, C.: Information Retrieval. Butterworths (1979). ISBN: 9780408709293
32. Walter, M., and Reussner, R.: Tool-based Attack Graph Estimation and Scenario Analysis for Software Architectures. In: European Conference on Software Architecture 2022 Tracks and Workshops (accepted, to appear)
33. Walter, M., *et al.*: Architectural Attack Propagation Analysis for Identifying Confidentiality Issues. In: International Conference on Software Architecture 2022
34. Walter, M., *et al.*: Architecture-based attack propagation and variation analysis for identifying confidentiality issues in Industry 4.0. at - Automatisierungstechnik 71(6), 443–452 (2023)
35. Walter, M., *et al.*: Dataset: Architecture-based Attack Path Analysis for Identifying Potential Security Incidents, `https://doi.org/10.5281/zenodo.7900356`
36. Yuan, B., *et al.*: An Attack Path Generation Methods Based on Graph Database. In: ITNEC, pp. 1905–1910 (2020)