Tool-based Attack Graph Estimation and Scenario Analysis for Software Architectures*

Maximilian Walter[0000−0003−0358−6644] and Ralf Reussner[0000−0002−9308−6290]

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

Abstract. With the increase of connected systems and the ongoing digitalization of various aspects of our life, the security demands for software increase. Software architects should design a secure and resistant system. One solution can be the identification of attack paths or the usage of an access control policy analysis. However, due to the system complexity identifying an attack path or analyzing access control policies is hard. Current attack path calculation approaches, often only focus on the network topology and do not consider the more fine-grained information a software architecture can provide, such as the components or deployment. In addition, the impact of access control policies for a given scenario is unclear. We developed an open-source attack propagation tool, which can calculate an attack graph based on the software architecture. This tool could help software architects to identify potential critical attack paths. Additionally, we extended the used access control metamodel to support a scenario-based access control analysis.

Keywords: Attack Propagation · Software Architecture · Security.

1 Introduction

Through the digitalization of our lives, more and more systems are connected with each other. This connection enables us to build smart systems and exchange data between different services. These connected systems should be resilient against cyber-attacks. One possibility to achieve this is by analyzing potential attack paths or access control policies.

However, estimating attack paths or analyzing access control policies is complicated. In advanced persistent threat (APT) [6], attackers often combine multiple security issues, such as vulnerabilities and access control properties, into one complex attack path. For instance, attackers often start with a phishing attack to get credentials and access to an element and then use this as a starting point to further attack the system [17]. This behavior can be reduced to first

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

getting access to a subsystem and then propagating further by exploiting different vulnerabilities. This pattern can also be seen in other incidents, such as in [34].

Existing attack path propagation approaches such as [9, 44] often only consider a network topology or a very reduced access control model. In contrast, other approaches with more fine-grained access control, such as Bloodhound [5] are very specific to the application domain for instance the Active Directory.

In addition, if an access control policy changes, it is often unclear what the impact for a specific scenario might be. A small policy change might lead to a too restrictive policy and blocks a legitimate and essential usage scenario, such as access to a machine in a production process. A policy change could potentially also have the opposite effect and be too open. This in turn enables malicious users to access data they should not be able to. Also, changes in the scenario are unclear. For instance, if the context of a user scenario changes, such as the access from a different location, it is unclear whether the user can still perform the scenario.

For solving these security issues, we developed an attack propagation analysis [40, 41] which uses a software architecture together with a fine-grained access control model to estimate potential attack paths. Software architects annotate vulnerabilities and access control policies to architectural elements and specify an attacker. The attacker contains the initial starting point, the capabilities (the attacks they can perform), and the knowledge (known credentials) of an attacker. Based on the provided information, our analysis calculates an attack graph. Software architects can use this attack graph to identify and evaluate potential attack paths. In addition, we extended the developed access control metamodel to support a scenario-based access control analysis. It enables architects to analyze certain intended usage scenarios against the software architecture together with access control policies and decide whether the scenarios are possible or not.

We will present our open-source Eclipse plugins[1] for the attack propagation and our short demonstration video[2]. We introduce our running example in Section 2. In Section 3, we describe the features of our original tool and in Section 3.3 describe the results of our tool. Afterward, we introduce our newly added scenario analysis in Section 4 and Section 5. The evaluation for our extension can be found in Section 6. Related work is described in Section 7. Section 8 concludes the paper and describes our future work.

2 Running Example

Figure 1 illustrates our running example. It is based on a scenario from a previous research project with industrial partners [2] that we extended in previous work [40] by adding vulnerabilities and access control information. It is settled in an Industry 4.0 environment.

In the scenario, a manufacturer (M) has a `Machine`. The `Machine` stores its data at the `ProductionDataStorage` which is deployed on the `StorageServer`.

[1] `https://fluidtrust.github.io/attack-propagation-doc/`
[2] `https://youtu.be/wiefWdTO9lo`

The `Machine` can be accessed by a `Terminal`, deployed on the `TerminalServer`. The `Terminal` is used by a technician from the service contractor (S). Because the machine data might contain sensitive data, S can only access the terminal during an incident such as a broken machine. Besides the machine part, there is the `ProductStorage`, which contains sensitive data about the product, such as blueprints. Therefore, S should have no access to this data. The `LocalNetwork` connects the different devices. A user with the attribute `Admin` has access to the `StorageServer` and `TerminalServer`. The `TerminalServer` is vulnerable to `CVE-2021-28374` [29]. As stated in the vulnerability's description, `CVE-2021-28374` can leak credentials.

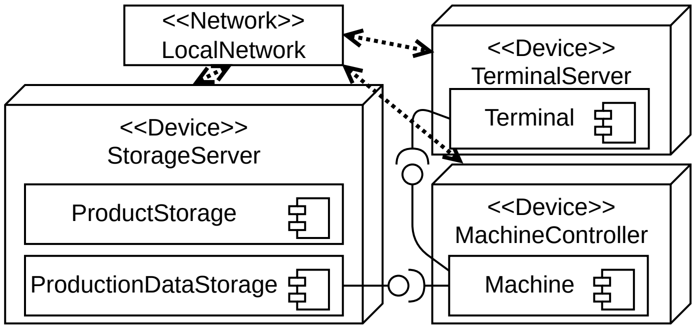


Fig. 1. Simplified architecture overview running example based on [40]

3 Attack Propagation Tool

Our attack propagation tool consists of three features, the software architecture modeling, an extension to model vulnerabilities and access control properties, and the attack propagation analysis.

3.1 Modeling Software Architecture

Our tooling is based on the existing Palladio-Bench, which extends the Eclipse Modelling Edition with Palladio Component Model (PCM) [31] specific editors and analyses. The Palladio-Bench is open-source and freely available. We also provide our extensions and the analysis freely as an open-source project. PCM is an Architecture Description Language (ADL) for the component-based development process. The modeled software architectures can be analyzed in various quality analyses such as performance [31] or confidentiality [33].

In PCM, software architects model the different aspects of the architecture in different models. In the `repository` model, a software architect can specify the components and their required and provided interfaces. The interfaces specify services with parameters and return values. Components can implement these services or delegate them to other services. These implemented services are called ServiceEffectSpecification (SEFF). The repository also contains datatype specifications necessary for the services. For our running example, we would specify the interfaces and the components there. For instance, for the `Machine`, we would specify a component *Machine* which provides the services for the `Terminal` and requires the services provided by `ProductionDataStorage`.

These specified components are then combined in the `system` model. Here, the different components are instantiated and wired together. The instantiated components are called *AssemblyContext*. For instance, in our running example, we instantiate the components and wire them together, e.g., the instantiated `Machine` is connected with the instantiated `Terminal`.

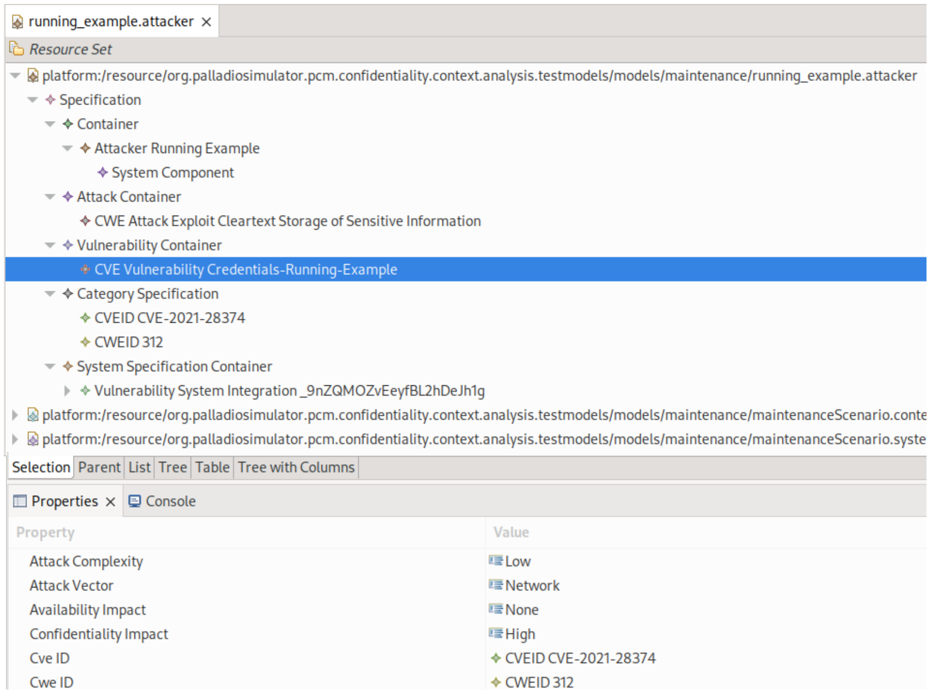
The different hardware elements are modeled in the `resourceenvironment` model. It contains *ResourceContainers* for processing nodes such as servers or notebooks and *LinkingResources* for network elements such as switches or routers. It also contains the link between them. In our running example, all elements with `<<Device>>` or `<<Network>>` would be in the resource environment.

The deployment of the different *AssemblyContexts* on the *ResourceContainers* is modeled in the `allocation` model, for instance, in our running example, the deployment of the `Terminal` on the `TerminalServer`. It, therefore, connects the resource environment with the system model. The usage model models the aggregated user behavior.

3.2 Attack Propagation Modeling

Our attack propagation extension [40] extends the existing PCM model by allowing to specify vulnerabilities and access control policies for certain architectural elements. In our case, these elements are *BasicComponents* as components in PCM, *AssemblyContexts*, *ServiceSpecifications*, *ResourceContainers*, and *LinkingResources*. In our tool, the access control properties are modeled in the `context` model. The access control specification is based on the eXtensible Access Control Markup Language (XACML) [43] standard for access control policy specification. We extended some elements such as the matching and attribute selection for easier modeling in PCM (see Figure 4). Software architects can specify with our extension access control policies and the attributes used in the access control decision. This allows us to support Attribute-based Access Control (ABAC) [15] for our access control policies. Reusing parts of XACML is beneficial since software architects or security experts might already be familiar with the standard and can therefore reuse their knowledge.

The vulnerability and attack specification is modeled in the `attacker` model. The vulnerability specification is based on the commonly used vulnerability classifications Common Weakness Enumeration (CWE) [26], Common Vulnerabilities and Exposure (CVE) [25], and Common Vulnerability Scoring System (CVSS) [8]. These classifications are often used by vendors or vulnerability databases such as the US. National Vulnerability Database (NVD) [30]. This allows software architects to reuse the existing knowledge about vulnerabilities. For instance, our



running_example.attacker

Resource Set

- platform:/resource/org.palladiosimulator.pcm.confidentiality.context.analysis.testmodels/models/maintenance/running_example.attacker
  - Specification
    - Container
      - Attacker Running Example
        - System Component
    - Attack Container
      - CWE Attack Exploit Cleartext Storage of Sensitive Information
    - Vulnerability Container
      - CVE Vulnerability Credentials-Running-Example
    - Category Specification
      - CVEID CVE-2021-28374
      - CWEID 312
    - System Specification Container
      - Vulnerability System Integration _9nZQMOZvEeyfBL2hDeJh1g
- platform:/resource/org.palladiosimulator.pcm.confidentiality.context.analysis.testmodels/models/maintenance/maintenanceScenario.conte
- platform:/resource/org.palladiosimulator.pcm.confidentiality.context.analysis.testmodels/models/maintenance/maintenanceScenario.syste

Selection | Parent | List | Tree | Table | Tree with Columns

Properties | Console

Property	Value
Attack Complexity	Low
Attack Vector	Network
Availability Impact	None
Confidentiality Impact	High
Cve ID	CVEID CVE-2021-28374
Cwe ID	CWEID 312

Fig. 2. Overview of the attacker model editor

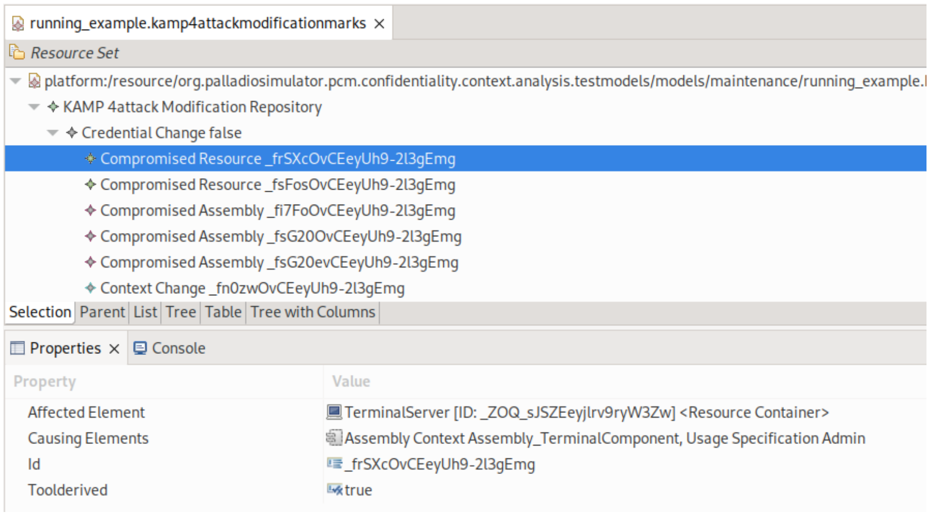
running example contains the vulnerability `CVE-2021-28374` [29]. Based on the description in the NVD, the software architects can know that credentials could be leaked and that the vulnerability can be exploited remotely. This information can then be added to our model (see also Figure 2) and is considered in the attack propagation.

Besides specifying the software architecture, access control policies and vulnerabilities, our analysis also needs the attacker as input. For the attacker, we specify the starting point in the software architecture, the capabilities and the knowledge about access properties. The starting point in the architecture can be an *AssemblyContext*, *LinkingResource* or *ResourceContainer*. The capabilities are the type of attacks an attacker can perform, i.e., the vulnerabilities they can exploit. Here, we also reused the CVE and CWE concept. For instance, the vulnerability in our running example can be exploited by CWE-312 [27] since `CVE-2021-28374` is part of CWE-312. Therefore, an attacker with the capability CWE-312 can exploit this vulnerability. The attributes used in the access control specification are knowledge about access properties. For instance, if an attacker in our running example has the `Admin` attribute, they could access the `StorageServer` and `TerminalServer`.

All these models can be edited with editors integrated into Eclipse. Figure 2 shows an overview of the attacker model editor. An overview of the new model elements can be found in our previous publication [40]. This model shows the `attacker` model for our running example. It shows the attacker (`Attacker Running Example`), the attack, the vulnerability (selected element), the CWE/CVE specification `Category Specification` and the integration into PCM (`System Specification Container`). For the selected vulnerability (blue), the editor shows additional properties (bottom part). These are properties for the vulnerabilities such as a `Low` attack complexity.

3.3 Attack Propagation Analysis & Tool Results

Our analysis [40] then calculates an attack path from one starting point and returns a list of all affected architectural elements. The attack path is calculated iteratively until no new element can be affected. It works internally similar as the KAMP [14] approach. However, they focussed on change propagation for maintenance, and we focus on attack propagation.



```
running_example.kamp4attackmodificationmarks
Resource Set
platform:/resource/org.palladiosimulator.pcm.confidentiality.context.analysis.testmodels/models/maintenance/running_example.
  KAMP 4attack Modification Repository
    Credential Change false
      Compromised Resource _frSXcOvCEeyUh9-2l3gEmg
      Compromised Resource _fsFosOvCEeyUh9-2l3gEmg
      Compromised Assembly _fi7FoOvCEeyUh9-2l3gEmg
      Compromised Assembly _fsG20OvCEeyUh9-2l3gEmg
      Compromised Assembly _fsG20evCEeyUh9-2l3gEmg
      Context Change _fn0zwOvCEeyUh9-2l3gEmg
Selection | Parent | List | Tree | Table | Tree with Columns

Properties | Console
Property              Value
Affected Element      TerminalServer [ID: _ZOQ_sJSZEeyjlrv9ryW3Zw] <Resource Container>
Causing Elements      Assembly Context Assembly_TerminalComponent, Usage Specification Admin
Id                    _frSXcOvCEeyUh9-2l3gEmg
Toolderived           true
```

Fig. 3. Overview analysis output

The results of our attack propagation analysis are stored in the `kamp4attack-modificationmarks` model. An excerpt is shown in Figure 3. In our case, it contains the attack path from the `Terminal` to the `StorageSever` by exploiting the vulnerability of the `TerminalServer` and getting the `Admin` credential. For instance, the selected element (blue) is the `TerminalServer`. In the properties view (bottom part) the software architect can see the `Affected Element` (here `TerminalServer`) and the reason (`Causing Effect`). Additionally, it contains an `ID` and the `Toolderived` property. The last property indicates that the analysis decided that the element is affected and that it was not one of the initial elements.

Besides this detailed list overview, our tooling can create a graph overview, where the connection between different elements is easier to recognize.

These analysis results can now be used by software architects. They have a list of affected elements and the reason. They can use this information to break potential attack paths. This could be done, for instance, by introducing mitigation approaches such as changing the credentials or updating the system. Afterward, they could remove the vulnerability from the attacker model and reanalyze the system to find out whether this mitigation approach solves the problem. However, the analysis could also be used to make trade-off decisions. For instance, a mitigation approach could be very costly either in system performance or monetary value. Here, architects could analyze different models with and without the mitigation and decide whether the risk is acceptable or whether they should choose the more secure system regardless of the cost. While our analysis cannot derive the other quality metrics, PCM already has support for various quality aspects like performance or cost [31] and the modeled system could be reused.

4 Modeling Architectural Access Control

As described in Section 3.2, our attack propagation tool uses an access control model based on XACML. So far, we have not investigated a scenario-based analysis for usage and misusage scenarios. In contrast to the attack propagation, we focus on verifying whether modelled usage scenarios are possible with the given software architecture and access control policies. This is especially important in evolutionary settings, where policies or scenarios change and system architects want to identify, whether the intended scenario is possible with the given architecture and access control policies.

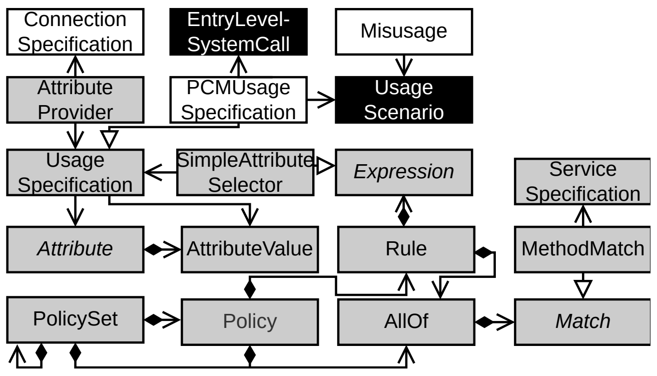


Fig. 4. Simplified access control policy metamodel with PCM integration

The access control metamodel (see Figure 4) contains mostly reused elements (gray) from our attack propagation tool. However, we needed to add some new elements (white) to annotate the PCM usage scenario description with context attributes. The black elements (`UsageScenario, EntryLevelSystemCall`) are original PCM elements. We first describe how a policy is specified and afterwards how we can assign attributes to a usage scenario.

All access control policies are contained in a `PolicySet`. It can consist of multiple `PolicySets` and `Policy` elements. A `Policy` can have multiple `Rule` elements. A `Rule` contains the actual access condition (as the `Expression`) and the result such as Permit or Deny. The `AllOf` specifies the target for which architectural elements the `Rule` or `Expression` should be applied. Because the target definition allows multiple `Rule` elements for one architectural element, there could exist multiple different access decisions. Therefore, the `PolicySet` and `Policy` have combination algorithms to reduce the decisions to one (see [43]). For a `Rule` or a `Policy` also multiple `AllOf` elements can be assigned. These multiple elements build a logical disjunction. Each `AllOf` element has multiple `Match` elements building a logical conjunction. The `Match` consists of boolean operations. For the `Match` we extended the XACML with custom match elements to better support the integration into PCM. `MethodMatch` is used to identify instantiated services with the `ServiceSpecification` element. PCM currently does not support the differentiation of instantiated services natively, therefore we added the `ServiceSpecification` element. It is a mapping between a service and the instantiated component. The `Expression` models the condition. The `SimpleAttributeSelector` wraps a simple attribute comparison. This eases the specification of a simple attribute condition. `UsageSpecification` is a tuple of `Attribute` and its concrete value as `AttributeValue`. For instance, in our running example, the role is a `Attribute` and the concrete role "technician" is a `AttributeValue`. `Attribute` elements can have references to certain architectural elements. This reference is later important for the model transformation to select the correct origin of an attribute.

For our running example, a policy could be as described in Listing 1.1. For simplicity reason, we only use one `Rule` and left out the PCM integration part. The combination algorithm is `Deny_Unless_Permit`, which states that every request is denied, unless there is a rule with a permit result. The `Rule` "Technician with Machine failure" has an expression that states that the requestor needs the role technician. In addition, other attributes can be added. The `AllOf` states that it targets a method. In the commented part, the correct element is selected.

Besides modeling the access control policy, we also need to model the user behavior and the user's context. In our previous publication [40], we only modeled the attacker and not normal users. Additionally, the behavior of the attacker was only modeled implicitly through the attack propagation, and the context was automatically derived from attacks. Here, we explicitly model the intended usage and the system context during a scenario.

The usage scenario in PCM describes the usage of the system during a specified scenario and it contains the `EntryLevelSystemCall` elements, which describe the called services by a user. In our approach, we enrich the `UsageScenario` with context information. The context is in our case the attributes used in the access

```
PolicySet {
  combining:Deny_Unless_Permit
  Policy{
    combining:Deny_Unless_Permit
    Rule{
      name:"Technician with Machine failure"
      decision:permit
      Expression{
        And{
          SimpleAttributeSelector{
            UsageSpecification{
              attribute:role
              value:technician
          }}/* further attribute conditions*/
      }}
      AllOf{
        MethodMatch{ /* integration into pcm */ }}}}}
```

Listing 1.1. Simplified textual representation of technican policy for services

control decision. The attributes are defined by the `PCMUsageSpecification`. Besides setting the context for the whole scenario, we also allow architects to specify the context for each `EntryLevelSystemCall`. This allows architects to specify context changes within a scenario. The context of an `EntryLevelSystemCall` overwrites the context of the `UsageScenario`.

Additionally, architects can specify context changes within a service. This is useful for internal access control. For instance, in our running example, the `Machine` retrieves the log data from the `ProductionDataStorage`. This action can happen on the user level, such as that the requestor is forwarded (here technician), or it can be done on the system level. In the last case the `Machine` would be the requestor at `ProductionDataStorage`. For modeling this context change, we added the `ConnectionSpecification` together with the `AttributProvider`. The `AttributProvider` is used in our attack propagation analysis for modeling that architectural elements can provide attributes. Here, we extended it with the `ConnectionSpecification`. This enables architects to specify that the context changes for certain connectors between services.

Besides modeling the intended usage with the usage scenario, we also enable the modeling of misusage scenarios. Misusage scenarios are based on misuscases [36] and mal-activity diagrams [35]. The idea is that architects can model scenarios that should not be possible. For instance, a possible misusage scenario for our running example could be that the technician gets access to the terminal without the machine in the failure state. In our case, architects can model this scenario by reusing the concepts of the normal usage scenario. However, in addition to modeling the scenario, they can create a `Misusage` element and assign an `UsageScenario`. In this case the `UsageScenario` is considered as a misusage scenario.

5 Analyzing Access Control Policies

After specifying the misusage and usage scenarios, and the access control policies, we can analyze each scenario for access control violations. Software architects can use these violations to change the access control policies or adapt the scenario to remove the violations. Our analysis consists of two steps. The first step is transforming our architectural access control model into a valid XACML file. We define valid, here, as valid according the XSD schema definition for the XACML 3.0 standard [43]. We can reuse the transformation from the attack propagation.

The second part is the scenario analysis. The scenario analysis first determines the context for each `UsageScenario`. Then it identifies the context for each `EntryLevelSystemCall` of a scenario. This context is either derived from the scenario when the call does not specify its own context or uses the specified context for the `EntryLevelSystemCall`. Then the analysis follows the call hierarchy of the services for each system call. There, it uses the `ExternalCall` from the SEFF to first identify the service type and then use the system model to determine the instantiated service. For identifying the instantiated service, we use the `AssemblyConnector`s of PCM. The analysis compares them together with the service type to the `ConnectionSpecification` elements of the `AttributeProvider` elements. If there exists at least one matching, it replaces the context, with the context for the `ConnectionSpecification`.

We then generate a valid XACML request out of the context for each instantiated service call and query a Policy Decision Point (PDP). The PDP evaluates the request based on the loaded policies and returns the access decision. This decision is then saved together with the instantiated service. During the request generation, we assign the `UsageSpecification` from the context to the three categories (subject, environment, resource) XACML uses and assign if necessary from which entity they come from. In XACML, this is called issuer.

After evaluating all access decisions for each scenario, the analysis decides whether a scenario is marked in the output as passed or not. For normal scenarios, the analysis marks a scenario as passed if every service call in it is permitted. For misusage scenarios, the analysis marks a scenario as passed if at least one service called is denied.

6 Evaluation

The evaluation of our attack propagation tool can be found in Walter et al. [40]. For the new scenario analysis, we describe the evaluation here. We follow the goal question metric approach [3]. Our evaluation goal is the *functional correctness* of our analysis. The analysis results should be correctly derived from the input model. Our evaluation questions are: **Q1** *Can the analysis determine the correct access decision for service calls?* **Q2** *Can the analysis determine the correct decision from usage scenarios or misusage scenarios?*

Question **Q1** focuses on the access decision for a service call. Here, we want to investigate whether every access decision for every service call is correct.

This question is important since we later use the results of the access decision to determine whether a scenario is passed. **Q2** investigates whether, based on the access decisions, the correct result for a scenario or misusage scenario is determined. This is important since these are the actual output results. For **Q1** and **Q2**, we use the Jaccard Coefficient (JC) [23] defined as $JC(A, B) = \frac{|A \cap B|}{|A \cup B|}$. It is used to compare the two sets A and B for equalness. The value range is from 1.0 for two equal sets to 0.0 for no intersection. The metric is used also in other design-time approaches in the PCM context such as [13]. The JC does not consider the order of elements. However, in our case we do not need the order (see Section 6.1).

6.1 Evaluation Design

In our evaluation, we use four case studies since case studies might provide better insights, show applicability, and increase the comparability between different approaches. We first describe the design for the different evaluation questions and introduce our used case studies afterwards.

For answering **Q1** and **Q2**, we manually create reference outputs for each case study. Afterwards, we compared the reference outputs against the results of the analysis. For **Q1** we see the access control decision as a set of tuples consisting of the access decision, the scenario, the connector, the intended service, the origin service and the involved instantiated component. These tuples are independent of each other, describe the access decisions and their order is irrelevant.

The scenario decision can also be seen as a tuple consisting of the scenario and the decision whether it is passed. The actual order of the scenarios is also not important for this tuple. Therefore, we can apply the JC and answer **Q2**. For the scenario analysis, we made sure that we had at least one normal scenario and one misusage scenario based on the description of the case study.

The first case study is the confidentiality case study TravelPlanner [20]. It is used in different confidentiality analyses, such as [22, 40]. We base our model on the model from [22]. The TravelPlanner describes the process of booking a flight from a mobile application. The main goal is that the credit card used for booking data must be explicitly declassified before the other entities can use it. In our scenario, we modelled this by adding an attribute for the classification.

Our second case study is based on the education example from Margrave [11]. We created, based on their description, a simple architecture model and created usage and misusage scenarios. The access control policies are also based on their description and their dataset. We need to adapt them since their policies are written for an older XACML version and do not contain the PCM references.

The third case study is based on the ABAC Banking case from [33]. It describes a simple banking system with branches in different regions. For each region, only the manager is allowed to handle celebrity customers. Regular customers are handled by a clerk. For our evaluation, we slightly adapted the access control model from a dataflow-based definition to a service-based definition since our approach only works on the service level. Additionally, we investigated the scenarios for the clerk and manager handling customers in the US branch and a misusage scenario where the clerk tries to handle a celebrity customer.

Our fourth case study is our running example. We investigate the scenario described in Section 2, scenarios regarding the saving of log data, accessing the product storage component. As a misusage scenario we investigated, that the technician tries to access the data without a machine failure.

6.2 Results & Discussion

For **Q1** and **Q2**, we achieve a JC from 1.0 for every case study. These are perfect results. These results are perfect since the case studies are small, and we only consider the decision (either access or pass). This simplifies the result. These results mean that for every scenario, our expected reference set is equally comparable to the result of the analysis. This indicates that our analysis works correctly. Based on this, architects could use our approach for analyzing the access control decisions for different scenarios. This enables them to analyze different alternative scenarios with various access control policies and see possible results. This analysis could help to harden the system by defining stricter access control policies and evaluating whether these stricter policies would still enable the use of the system in a certain scenario. Additionally, it can help in policy changes to not forget malicious scenarios by explicitly modelling them as misusage scenarios and analyzing them. This can help to prohibit malicious usage through policy changes, because the misusage scenario can automatically check for violations after a policy change.

6.3 Threats to Validity

Based on the guidelines for case study validity from Runeson et al. [32], we categorize our threats to validity into four categories.
Internal Validity: This discusses that only the expected factors influence the results. Because of the different input models and that we evaluate only the result, our evaluation highly depends on the used models. Even more, we manually created the output models. We try to lower the risk by using mostly external case studies and deriving the expected results based on their descriptions. Another threat is the size of the models since the models are quite small. However, they already cover our approach's important functionality, such as the context derivation, transformation of the access control model, access control decisions, and misusage and scenario analysis. Therefore, adding more architectural elements might increase the number of result objects but would not gain more insights. Hence, we assume the risk to be low.
External Validity: This is how useful the results are for other researchers and, therefore, how generalizable the results are. Using a case study based evaluation, we might increase the insights into the problem, but the case study might be not representative. Therefore, we choose mostly external case studies, which are used in various approaches such as [20, 22, 40]. Additionally, the maintenance case study (the running example) is based on a scenario from our industrial

partners in a previous research project [2]. These external case studies and the industrial one lower the risk of the representativeness. However, the results so far only indicate the functional correctness of the analyses and not the correctness of the approach in general. Therefore, we plan to address this in the future and investigate further case studies and scenarios.
Construct Validity: In our case, this is whether the metrics are appropriate for the intended goal. For **Q1** and **Q2**, we use the JC. It is also used in similar approaches, such as [13]. Its main restriction is that it cannot differentiate the order between elements. However, in our cases, as discussed in Section 6.1, the order is not relevant. Additionally, for **Q1** and **Q2**, we consider correctness as that the analysis output is equal to the reference output and this is the intended goal for the metric. Therefore, we consider the risk for the metrics to be low.
Reliability: This describes whether other researchers can reproduce the results later. By using statistical metrics we avoid subjective interpretation and therefore can increase reproducibility. Additionally, our dataset [42] allows other researchers to verify the results.

6.4 Limitations

For our approach, we need an architectural model to annotate the system policies. While this is not always true, there exist reengineering approaches, such as Kirschner et al. [21] , which help to create one from existing software.

Regarding the dynamic changes of context attributes, we specify that they are at least foreseeable during design time so that they can be expressed in the policies. This is similar to our definition of dynamic changes [39]. Context attributes or scenarios which are not considered during runtime cannot be analyzed.

7 Related Work

We list related approaches regarding our attack propagation in Walter et al. [40]. Here, we focussed on approaches regarding the scenario analysis. We categorized the related work in access control models, access control policy analyses and model-driven confidentiality analyses.
Access Control Models: Role-based access control (RBAC) [10] considers the role for the access decision. However, usually, the role is the only context that is considered. Organisation-based access control (OrBAC) [19] was developed for complex access control policies within an organization and supports multiple different contexts [7]. However, the industrial application is currently very limited. ABAC [15] also considers the context for access decision. ABAC is often described as a dynamic access control approach. XACML [43] is an implementation of ABAC.
Access Control Policy Analyses: Jabal et al. [16] analyze various policy analysis approaches. Margrave [11] is a XACML based verification and change-impact analysis. It uses binary decision trees to decide whether a user can perform certain operations or determine the impact of a policy change. In contrast to our

approach, they do not consider the software architecture or misusage scenarios. Alberti et al. [1] analyze a modified RBAC model with additional properties, which can be seen as context properties. One analysis aspect from them is the delegation of RBAC policies. Our approach does not consider the delegation, but we support different scenarios and misusage based on the software architecture. Another XACML based analysis is developed by Turkmen et al. [38]. They internally use satisfiability modulo theories (SMT) to analyze different access control properties such as a change impact and attribute hiding [38]. Overall, there exists different policy analyses approach. However, so far they do not support the scenario-based analysis based on the software architecture. Nevertheless, the XACML based approaches can be used in combination with our access control analysis since XACML files could be used as a universal exchange format.

Model-driven Confidentiality Analyses Various approaches for model-driven confidentiality analyses exist [28]. We focus here on the most relevant for us. UMLsec [18] is a security extension for UML. It can analyze various security properties such as secure exchange and secure communication. However, they do so far not consider a fine-grained access control model as it is necessary for our running example. Another security extension for UML is SecureUML [24]. It uses an RBAC policy model that can be extended by OCL statements to support context properties. Additionally, it also supports an automatic policy analysis [4]. In contrast, we can consider misusage scenarios in our analysis, and our modeling closely follows an industrial standard which eases the modeling. Data-centric Palladio [33] is a dataflow-based security analysis for the Palladio approach [31]. It provides different analyses such as information flow or access control. However, our extension is defined on the service declaration and not on data objects, and they do not support misusage scenarios. Another dataflow information flow analysis is SecDFD [37]. In contrast, we support misusage scenarios and work on the software architecture. Gerking et al. [12] present a confidentiality analysis based on timed automatons to analyze the real-time properties of a system. The iFlow approach [20] is a confidentiality analysis for information flow by using UML profiles. In contrast to both previous mentioned ones, we focus more on access control and misusage scenarios.

8 Conclusion & Future Development

In this paper, we first presented our tool for an attack propagation [40]. The tool can help software architects to build more secure systems by providing potential attack paths, which can be broken by introducing mitigation strategies. Secondly, we introduced our approach for a scenario-based access control analysis based on the software architecture. It extends, the access control metamodel from our attack propagation tool and enables software architects to analyze the intended usage and misusage scenarios against the modeled software architecture and the specified software architecture. Our evaluation indicates, that we can detect access violations for system calls and deduce whether scenarios are possible based on the access control decision.

In the future, we want to extend our attack propagation approach by considering advanced mitigation strategies. Additionally, we plan to develop a new security analysis using our metamodel, such as an attack surface analysis and apply both our existing analyses in more case studies. Besides adding new functionality, we also plan to improve the documentation and usability. Here, our focus is on better editor support and error handling. Another open point is the scalability of our analyses. We want to investigate whether we can improve the runtime behavior for more extensive systems.

References

1. Alberti, F., *et al.*: Efficient symbolic automated analysis of administrative attribute-based RBAC-policies. In: ASIACCS, p. 165 (2011)
2. Al-Ali, R., *et al.*: Modeling of Dynamic Trust Contracts for Industry 4.0 Systems. In: ECSA-C. ACM (2018)
3. Basili, G., *et al.*: The goal question metric approach. Encyclopedia of software engineering (1994)
4. Basin, D., *et al.*: Automated analysis of security-design models. Information and Software Technology 51(5), 815–831 (2009)
5. BloodHound Enterprise, `https://bloodhoundenterprise.io/` (visited on 10/05/2021)
6. Cole, E.: Advanced persistent threat: understanding the danger and how to protect your organization. Newnes (2012)
7. Cuppens, F., and Miège, A.: Modelling contexts in the Or-BAC model. In: ACSAC, pp. 416–425 (2003)
8. CVSS SIG, `https://www.first.org/cvss/` (visited on 10/25/2021)
9. Deloglos, C., *et al.*: An Attacker Modeling Framework for the Assessment of Cyber-Physical Systems Security. In: Computer Safety, Reliability, and Security, pp. 150–163. Springer (2020)
10. Ferraiolo, D., *et al.*: Role-based access control (RBAC): Features and motivations. In: ACSAC, pp. 241–248 (1995)
11. Fisler, K., *et al.*: Verification and change-impact analysis of access-control policies. In: ICSE, p. 196 (2005)
12. Gerking, C., and Schubert, D.: Component-Based Refinement and Verification of Information-Flow Security Policies for Cyber-Physical Microservice Architectures. In: ICSA, pp. 61–70 (2019)
13. Heinrich, R.: Architectural runtime models for integrating runtime observations and component-based models. JSS 169, 110722 (2020)
14. Heinrich, R., *et al.*: Architecture-based change impact analysis in cross-disciplinary automated production systems. JSS 146, 167–185 (2018)
15. Hu, V., *et al.*: Attribute-Based Access Control. Computer 48(2), 85–88 (2015)
16. Jabal, A.A., *et al.*: Methods and Tools for Policy Analysis. ACM Computing Surveys 51(6), 1–35 (2019)
17. Johns, E.: Cyber Security Breaches Survey 2021: Statistical Release. Tech. rep., p. 66. Department for Digital, Culture, Media & Sport, UK and Ipsos Mori (2021)
18. Jürjens, J.: "UMLsec: Extending UML for Secure Systems Development". In: UML. Vol. 2460. Springer Berlin Heidelberg, 2002, pp. 412–425.
19. Kalam, A., *et al.*: Organization based access control. In: POLICY'03
20. Katkalov, K., *et al.*: Model-Driven Development of Information Flow-Secure Systems with IFlow. In: SOCIALCOM'13, pp. 51–56

21. Kirschner, Y.R., *et al.*: Automatic Derivation of Vulnerability Models for Software Architectures. In: ICSA-C (*accepted, to appear*) (2023)
22. Kramer, M., *et al.*: Model-Driven Specification and Analysis of Confidentiality in Component-Based Systems. Tech. rep. 12, KIT-Department of Informatics (2017)
23. Levandowsky, M., and Winter, D.: Distance between sets. Nature 234(5323), 34–35 (1971)
24. Lodderstedt, T., *et al.*: SecureUML: A UML-Based Modeling Language for Model-Driven Security. In: pp. 426–441. Springer (2002)
25. MITRE: CVE, `https://cve.mitre.org/` (visited on 10/25/2021)
26. MITRE: CWE, `https://cwe.mitre.org/` (visited on 10/25/2021)
27. MITRE: CWE-312, `https://cwe.mitre.org/data/definitions/312.html` (visited on 10/25/2021)
28. Nguyen, P., *et al.*: An extensive systematic review on the Model-Driven Development of secure systems. Information and Software Technology 68, 62–81 (2015)
29. NIST: CVE-2021-28374, `https://nvd.nist.gov/vuln/detail/CVE-2021-28374` (visited on 10/25/2021)
30. NVD, `https://nvd.nist.gov/vuln` (visited on 10/25/2021)
31. Reussner, R., *et al.*: Modeling and Simulating Software Architectures – The Palladio Approach. MIT Press, Cambridge, MA (2016)
32. Runeson, P., and Höst, M.: Guidelines for conducting and reporting case study research in software engineering. Empirical Software Engineering 14(2), 131 (2008)
33. Seifermann, S., *et al.*: Detecting Violations of Access Control and Information Flow Policies in Data Flow Diagrams. JSS 184 (2021)
34. Shu, X., *et al.*: Breaking the Target: An Analysis of Target Data Breach and Lessons Learned. arXiv:1701.04940 [cs] (2017)
35. Sindre, G.: "Mal-Activity Diagrams for Capturing Attacks on Business Processes". In: Requirements Engineering. Vol. 4542. LNCS. Springer, 2007, pp. 355–366.
36. Sindre, G., and Opdahl, A.L.: Eliciting security requirements with misuse cases. Requirements Engineering 10(1), 34–44 (2005)
37. Tuma, K., *et al.*: Flaws in Flows: Unveiling Design Flaws via Information Flow Analysis. In: ICSA, pp. 191–200 (2019)
38. Turkmen, F., *et al.*: Analysis of XACML Policies with SMT. In: Principles of Security and Trust, pp. 115–134. Springer (2015)
39. Walter, M., *et al.*: A Taxonomy of Dynamic Changes Affecting Confidentiality. In: 11th Workshop Design For Future - Langlebige Softwaresysteme (2020)
40. Walter, M., *et al.*: Architectural Attack Propagation Analysis for Identifying Confidentiality Issues. In: IEEE ICSA, pp. 1–12 (2022)
41. Walter, M., *et al.*: Architectural attack propagation in industry 4.0. at - Automatisierungstechnik (*accepted, to appear*) (2023)
42. Walter, M., *et al.*: Dataset: Palladio Context-based Scenario Analysis, (2022). `https://doi.org/10.5281/zenodo.7431562`
43. XACML, `https://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.html` (visited on 10/25/2021)
44. Yuan, B., *et al.*: An Attack Path Generation Methods Based on Graph Database. In: ITNEC, pp. 1905–1910 (2020)