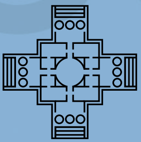
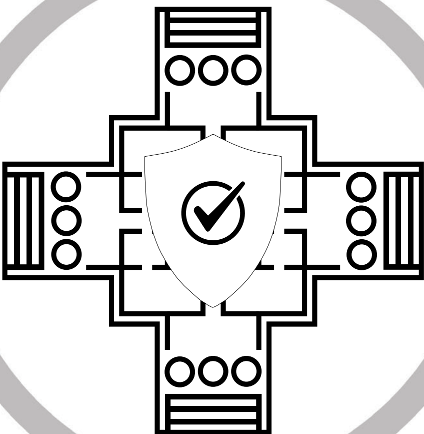
The Karlsruhe Series on
Software Design
and Quality

41



Context-based Access Control and Attack Modelling and Analysis

Maximilian Walter



Scientific
Publishing

Maximilian Walter

Context-based Access Control and Attack Modelling and Analysis

The Karlsruhe Series on Software Design and Quality
Volume 41

Dependability of Software-intensive Systems group
Faculty of Computer Science
Karlsruhe Institute of Technology

and

Software Engineering Division
Research Center for Information Technology (FZI), Karlsruhe

Editor: Prof. Dr. Ralf Reussner

Context-based Access Control and Attack Modelling and Analysis

by
Maximilian Walter

KIT Scientific Publishing

Karlsruher Institut für Technologie
KASTEL – Institut für Informationssicherheit und Verlässlichkeit

Context-based Access Control and Attack Modelling and Analysis

Zur Erlangung des akademischen Grades eines Doktors der Ingenieurwissenschaften von der KIT-Fakultät für Informatik des Karlsruher Instituts für Technologie (KIT) genehmigte Dissertation

von Maximilian Walter

Tag der mündlichen Prüfung: 6. Dezember 2023
1. Referent: PD. Dr. Robert Heinrich
2. Referent: Prof. Dr. Jan Jürjens

Impressum

Karlsruher Institut für Technologie (KIT)
KIT Scientific Publishing
Straße am Forum 2
D-76131 Karlsruhe

KIT Scientific Publishing is a registered trademark
of Karlsruhe Institute of Technology.
Reprint using the book cover is not allowed.

www.ksp.kit.edu

This document – excluding parts marked otherwise, the cover, pictures and graphs – is licensed under a Creative Commons Attribution-Share Alike 4.0 International License (CC BY-SA 4.0): https://creativecommons.org/licenses/by-sa/4.0/deed.en

The cover page is licensed under a Creative Commons Attribution-No Derivatives 4.0 International License (CC BY-ND 4.0): https://creativecommons.org/licenses/by-nd/4.0/deed.en

Print on Demand 2024 – Gedruckt auf FSC-zertifiziertem Papier

ISSN 1867-0067
ISBN 978-3-7315-1362-9
DOI: 10.5445/KSP/1000170265

Danksagung

Zuerst möchte ich mich bei meinem Betreuer und Gutachter PD. Dr. Robert Heinrich bedanken. Durch die etlichen Diskussionen mit ihm ist erst diese Arbeit entstanden. Er hat dabei immer Zeit für mich und meine Forschung gehabt. Dieses Feedback war in verschiedenen Projekten, wissenschaftlichen Publikationen und auch meiner Dissertation sehr hilfreich. Daneben möchte ich mich bei Prof. Dr. Ralf Reussner bedanken, in dessen Gruppe ich sein durfte. Ralf war neben Robert immer ein guter Ansprechpartner und lieferte wichtiges Feedback für diese Arbeit. Insbesondere hatte Ralf auch immer ein offenes Ohr für organisatorische und andere Anliegen für die Arbeit am Lehrstuhl. Dadurch war es möglich, dass wir am Lehrstuhl ein sehr gutes Arbeitsklima hatten und die tägliche Arbeit neben der Promotion auch immer Spaß gemacht hat.

Dazu gehören aber natürlich auch die anderen Kolleginnen und Kollegen an beiden Lehrstühlen. Ich möchte mich dementsprechend bei allen für die immer gute Zusammenarbeit und den Zusammenhalt bedanken. Insbesondere die Diskussionen in Doktrandenrunden, Forschungstreffen oder anderweitigen Treffen haben mir immer sehr geholfen. Ich möchte mich insbesondere bei den Kollegen Dr. Stephan Seifermann und Sebastian Hahner bedanken, mit denen ich in verschiedenen Forschungsprojekten arbeiten durfte. Insbesondere Stephan hat mich auch immer zu technischen Fragen wie Buildprozessen sehr gut beraten und ein offenes Ohr gehabt. Ich möchte mich auch bei allen meinen Bürokollegen, Nicolas Boltz, Sebastian Hahner, Kai Marquardt, Dr. Stephan Seifermann, Dominik Werle und Jan Wittler bedanken. Insbesondere bei Nicolas der zuerst als mein Student und anschließend als Kollege mich am Lehrstuhl begleitet hat.

Ein besonderer Dank geht auch an meinen Kolleginnen und Kollegen Dominik Fuchß, Angelika Kaplan und Yves Kirschner mit denen ich in der Übungsleitung in Programmieren zusammenarbeiten durfte. Dank ihnen hatte ich erst die Zeit meine Dissertation zu beenden.

Vielen Dank auch an alle anderen, die mich auf diesem Weg begleitet haben und mir immer zur Seite gestanden sind.

Zum Schluss möchte ich mich gerne noch bei meiner Familie bedanken. Insbesondere meinen Eltern und meinem Bruder, durch deren Unterstützung und permanente Ermutigung war es mir erst möglich zu studieren und auch meine Dissertation zu schreiben.

Abstract

In this thesis, we developed architectural security analyses to identify access violations and attack paths.

Through the ongoing digitalization and increasing networking between various aspects of our daily life, the significance of security is on the rise. System security encompasses multiple properties, including confidentiality and integrity. In our research, we specifically concentrate on confidentiality. The main aspect of a confidential system is that it only shares the required data with authorized entities. Unauthorized or malicious entities are prevented from gaining access to the data.

However, designing a confidential system is challenging due to numerous factors influencing whether a system can be deemed confidential. One significant influencing factor is the access control. The access control is specified by access control policies. These policies define the conditions under which access can be granted to each entity within a system. Due to the ongoing digitalization, these access control policies must increasingly consider context factors when accessing data. For instance, the context may include a user's time or location. As considering contexts becomes more prominent, the complexity of specifying access control policies also rises. Consequently, there is a greater likelihood of access control policy misspecifications. Therefore, context information plays a crucial role in determining the impact of access control policies. However, due to the complexity of these policies, assessing their impact is equally challenging. This challenging impact estimation makes analyses that consider the context to determine the impact necessary.

In addition to access control policies, vulnerabilities can significantly affect the confidentiality of a system. Attackers exploit these vulnerabilities to gain unauthorised access to protected entities within the system, thereby bypassing access control policies. Vulnerabilities not only grant direct access to entities but can also lead to the unauthorised disclosure of authorisation or credential information. Attackers can leverage this leaked information to

gain access to other entities. However, vulnerabilities are also dependent on access control systems, as certain vulnerabilities may require authorisation for exploitation. For example, some vulnerabilities can only be exploited by authorised users. Consequently, when estimating the impact of a vulnerability, an analysis must consider the access control properties. Furthermore, the context of the attacker is crucial, as some vulnerabilities can only be exploited if the attacker has previously compromised other entities within the system. This results in a chain of compromised entities, often referred to as attack paths. These paths consist of lateral movement and vulnerability chaining representing the multiple exploitations of vulnerabilities and access control policies through attackers. The automatic derivation of these potential attack paths can aid in estimating the impact on confidentiality by providing experts with feedback regarding potential compromised elements.

Existing approaches for estimating the security or impact of access control policies or vulnerabilities often concentrate solely on one aspect, either the access control policies or the vulnerabilities. Approaches considering both properties tend to be highly specialized, focusing on a single application domain, like Microsoft Active Directory, or employing a limited access control model. Moreover, many existing approaches primarily operate on a network topology, which aids in modelling but fails to account for additional important factors such as deployment and components within the system.

Software architecture models can provide this information. In addition, the usage of models enables us to analyse a system already during the development or in a downtime. Hence, it helps in achieving *Security by Design*. Our specific contributions are as follows: I) Development of an access control metamodel to specify context-based access control policies within the software architecture. II) Creation of a vulnerability metamodel to specify vulnerabilities within software architectures. III) Development of a scenario-based access control analysis to analyse access control policies and identify access violations. IV) Development of two attack analyses that generate attack paths using vulnerabilities and access control policies based on the software architecture. One analysis focuses on the propagation of attacks from a specific starting point in the software architecture, while the other identifies attack paths leading to specific architectural elements.

We evaluated our security analyses on different evaluation scenarios. These scenarios are derived from evaluation cases found in related work, as well as

real-world security incidents. For the first analysis, we investigated the accuracy of identifying access violations. Our findings indicate a high accuracy in this regard.

Regarding the two attack analyses, we investigated the accuracy in identifying compromised elements, the potential effort reduction through using our analyses and the scalability. Our findings indicate a high accuracy and an effort reduction. However, the scalability for both approaches could be improved. Nevertheless, for smaller software architectures, it is sufficient.

Our developed approach can help software architects to design secure systems. By providing access violations and attack paths, our approach helps in estimating the impact of access control policies and vulnerabilities within the software architecture. Moreover, through utilising software architecture models, our approach can provide feedback already during the design of the software. This can help to develop *Secure Software by Design*.

Zusammenfassung

In dieser Arbeit haben wir architekturelle Sicherheitsanalysen entwickelt, um Zugriffsverletzungen und Angriffspfade zu ermitteln.

Durch die fortschreitende Digitalisierung und die zunehmende Vernetzung steigt die Bedeutung der IT-Sicherheit. Die Sicherheit eines Systems besteht aus mehreren verschiedenen Eigenschaften wie Vertraulichkeit oder Integrität. In unserer Arbeit konzentrieren wir uns auf die Vertraulichkeit. Ein vertrauliches System teilt nur die benötigten Daten mit autorisierten Entitäten. Unbefugte oder böswillige Personen erhalten keinen Zugang zu vertraulichen Daten.

Die Entwicklung eines vertraulichen Systems ist jedoch schwierig, da viele verschiedene Eigenschaften Einfluss auf die Vertraulichkeit haben. Ein wichtiger Einflussfaktor ist die Zugangskontrolle. Zugriffskontrollrichtlinien definieren für jedes Element innerhalb eines Systems, unter welchen Bedingungen der Zugriff gewährt werden kann. Diese Zugriffskontrollrichtlinien berücksichtigen oft den Kontext für den Zugriff. Der Kontext kann z.B. die Zeit oder der Standort von Personen sein. Durch die Berücksichtigung steigt die Komplexität der Spezifikation der Zugriffskontrolle. Dies kann zu einer Fehlspezifikation führen. Daher ist es wichtig, die Auswirkungen einer Zugriffskontrollrichtlinie zu ermitteln. Aufgrund der Komplexität ist es jedoch schwierig, die Auswirkungen zu bestimmen, da die Analyse auch den Kontext berücksichtigen muss.

Neben Zugriffskontrollrichtlinien können auch Schwachstellen die Vertraulichkeit des Systems beeinflussen. Schwachstellen können von Angreifer:innen ausgenutzt werden, um Zugang zu geschützten Entitäten im System zu erhalten. Sie ermöglichen es den Angreifer:innen also, die Zugangskontrollrichtlinien zu umgehen. Schwachstellen ermöglichen nicht nur den direkten Zugang zu Entitäten, sondern ermöglichen Angreifer:innen auch die Berechtigung anderer Personen zuerlangen. Diese Berechtigung kann dann von Angreifer:innen verwendet werden, um sich bei anderen Elementen Zugang

zu verschaffen. Schwachstellen hängen jedoch auch von Zugangskontrollsystemen ab, da für einige Schwachstellen eine Berechtigung erforderlich ist. So können beispielsweise einige Schwachstellen nur von berechtigten Personen ausgenutzt werden. Um die Auswirkungen einer Schwachstelle abschätzen zu können, muss eine Analyse daher auch die Eigenschaften der Zugangskontrolle berücksichtigen. Darüber hinaus ist der Kontext der Angreifer:innen wichtig, da einige Schwachstellen nur dann ausgenutzt werden können, wenn der Angreifer:innen zuvor andere Entitäten im System kompromittiert haben. Daher wird bei Angriffen eine verkettete Liste kompromittierter Entitäten erstellt. Diese Liste wird auch als Angriffspfad bezeichnet. Sie besteht aus einer Kette von Schwachstellen, die die mehrfache Ausnutzung von Schwachstellen und Zugangskontrollrichtlinien durch Angreifer:innen darstellen. Die automatische Ableitung dieser möglichen Angriffspfade kann verwendet werden, um die Auswirkungen auf die Vertraulichkeit abzuschätzen, da sie den Expert:innen eine Rückmeldung darüber gibt, welche Elemente kompromittiert werden können.

Bestehende Ansätze zur Abschätzung der Sicherheit oder der Auswirkungen von Zugangskontrollrichtlinien oder Schwachstellen konzentrieren sich oft nur auf eine der beiden Eigenschaften. Ansätze, die beide Eigenschaften berücksichtigen, sind in der Anwendungsdomäne oft sehr begrenzt, z.B. lösen sie es nur für eine Anwendungsdomäne wie Microsoft Active Directory oder sie berücksichtigen nur ein begrenztes Zugangskontrollmodell. Darüber hinaus arbeiten die meisten Ansätze mit einer Netzwerktopologie. Dies kann zwar bei der Modellierung hilfreich sein, doch berücksichtigt eine Netzwerktopologie in der Regel keine weiteren Eigenschaften wie Bereitstellung von Diensten auf Servern oder die Nutzung von Komponenten.

Software-Architekturmodelle können diese Informationen jedoch liefern. Darüber hinaus ermöglicht die Verwendung von Modellen, ein System bereits während der Entwicklung oder während eines Ausfalls zu analysieren. Daher hilft es bei der Verwirklichung von „*Security by Design*". Im Einzelnen sind unsere Beiträge: I) Wir haben ein Metamodell für die Zugriffskontrolle entwickelt, um kontextbasierte Zugriffskontrollrichtlinien in der Software-Architektur zu spezifizieren. II) Zusätzlich haben wir ein Schwachstellen-Metamodell entwickelt, um Schwachstellen in Software-Architekturen zu spezifizieren. III) Die Zugriffskontrollrichtlinien können in einer szenariobasierten Zugriffskontrollanalyse analysiert werden, um Zugriffsverletzungen zu identifizieren. IV) Wir haben zwei Angriffsanalysen entwickelt. Beide können Angriffspfade auf einem Architekturmodell generieren und Schwachstellen und Zugangs-

kontrollrichtlinien verwenden. Die eine Analyse betrachtet die Angriffsausbreitung von einem bestimmten Startpunkt in der Software-Architektur. Die andere findet Angriffspfade, die zu einem bestimmten Architekturelement führen.

Wir haben unsere Sicherheitsanalysen anhand verschiedener Evaluierungsszenarien evaluiert. Diese Szenarien wurden auf der Grundlage von Evaluierungsfällen aus verwandten Arbeiten oder realen Sicherheitsvorfällen erstellt. Für die erste Analyse haben wir die Genauigkeit bei der Identifizierung von Zugriffsverletzungen untersucht. Unsere Ergebnisse deuten auf eine hohe Genauigkeit hin.

Für die beiden Angriffsanalysen untersuchten wir die Genauigkeit hinsichtlich der gefundenen kompromittierten Elemente, die Aufwandsreduzierung bei der Verwendung unserer Analysen und die Skalierbarkeit. Unsere Ergebnisse deuten auf eine hohe Genauigkeit und eine Aufwandsreduzierung hin. Allerdings ist die Skalierbarkeit für beide Ansätze nicht ideal. Für kleinere Software-Architekturen ist sie jedoch akzeptabel.

Der von uns entwickelte Ansatz kann Software-Architekt:innen dabei helfen, sicherere Systeme zu entwerfen. Der Ansatz kann die Auswirkungen von Zugriffskontrollrichtlinien anhand von Zugriffsverletzungen und für Schwachstellen zusammen mit Zugriffskontrollrichtlinien anhand von Angriffspfaden aufzeigen. Durch die Verwendung von Software-Architekturmodellen kann unser Ansatz dieses Feedback bereits während des Entwurfs der Software liefern. Dies kann helfen, nach „*Security by Design*“ zu entwickeln.

Contents

Danksagung . . . . **i**

Abstract . . . . **iii**

Zusammenfassung . . . . **vii**

List of Figures . . . . **xvii**

List of Tables . . . . **xxi**

List of Acronyms . . . . **xxiii**

I. Prologue **1**

1. Introduction **3**
- 1.1. Motivation 3
- 1.2. Problem Statement 9
- 1.3. Research Questions 11
- 1.4. Contributions 13
- 1.5. Outline 16

2. Foundations **17**
- 2.1. Software Architecture Description and Analysis 17
  - 2.1.1. Model-Driven Software Development 17
  - 2.1.2. Software Architecture and Palladio Component Model 20
  - 2.1.3. Karlsruhe Architectural Maintainability Prediction Approach 21
- 2.2. Security Related Concepts and Terms 22
  - 2.2.1. Access Control Specification 22
  - 2.2.2. Security Incidents and Vulnerabilities 24
  - 2.2.3. Vulnerability Classification 25

2.2.4. Misuse Case 26
2.3. Categorising Threats to Validity 26

3. Running Example 29

II. Contributions 35

4. Modelling Influencing Factors for Context-Based Security 37
4.1. Identification of Services and Components 38
4.2. Considering Access Control Properties in Software Architecture Models 41
4.2.1. Requirements for Modelling Access Control Properties 42
4.2.2. Modelling Attributes 45
4.2.3. Modelling Access Control Policies 48
4.2.4. Modelling Attribute Providers and Scenarios 54
4.2.5. Transformation to XACML & Access Requests 56
4.3. Considering Vulnerabilities in Software Architecture Models . 67
4.3.1. Requirements for Modelling Vulnerabilities 68
4.3.2. Modelling Identifiers for Vulnerabilities 71
4.3.3. Modelling Vulnerabilities 71
4.3.4. Integration in Palladio 79
4.3.5. Automatic Derivation of Vulnerabilities 81
4.4. Considering Attacks in Software Architecture Models 82
4.4.1. Requirements for Modelling Attacks 83
4.4.2. Modelling Attacks 84
4.5. Considering Attackers in Software Architecture Models 85
4.5.1. Modelling Attackers for Attack Propagation 86
4.5.2. Modelling Attackers for Filtered Attack Paths 87

5. Analysing Software Architectures for Potential Security Incidents . 93
5.1. Scenario-Based Access Usage Analysis 94
5.1.1. Process for Analysing Scenario-based Access Control Policies 95
5.1.2. Analysing Scenarios for Access Violations 99
5.1.3. Analysing Misusage Scenarios for Access Violations . 103
5.1.4. Result Model for the Access Usage Analysis 104

5.2. Attack Propagation Analysis 106
5.2.1. Process for Analysing Attack Propagations with the Software Architecture 108
5.2.2. Attack Propagation Process 110
5.2.3. Data Extraction 111
5.2.4. Analysing Attack Propagations 112
5.2.5. Result Model for Attack Propagation 126
5.3. Targeted Attack Graph Analysis 129
5.3.1. Process for Analysing Attack Graphs based on the Software Architecture 130
5.3.2. Attack Graph Analysis Process 131
5.3.3. Creating an Attack Graph 131
5.3.4. Identifying Attack Paths 136
5.3.5. Result Model for the Targeted Attack Graph Analysis 142

III. Validation **145**

6. Evaluation Scenarios 147
6.1. TravelPlanner 148
6.2. Power Grid 152
6.3. Target 156
6.4. Cloud Infrastructure 160
6.5. ABAC-Banking 162
6.6. Education 163
6.7. Maintenance Scenario 165

7. Evaluation 169
7.1. Usage Analysis 170
7.1.1. Goal, Question, Metric 170
7.1.2. Evaluation Design 173
7.1.3. Results & Discussion 175
7.1.4. Threats to Validity 177
7.2. Attack Propagation 181
7.2.1. Goal, Question, Metrics 181
7.2.2. Evaluation Design 188
7.2.3. Results & Discussion of Accuracy 193
7.2.4. Results & Discussion of Effort Reduction 195
7.2.5. Results & Discussion of Scalability 203

7.2.6. Threats to Validity 205
7.3. Targeted Attack Graph Analysis 211
7.3.1. Goal, Question, Metrics 211
7.3.2. Evaluation Design 216
7.3.3. Results & Discussion of Accuracy 220
7.3.4. Results & Discussion of Effort Reduction 221
7.3.5. Results & Discussion of Scalability 223
7.3.6. Threats to Validity 224
7.4. Assumption and Limitations 230
7.5. Overall Evaluation Results & Discussion 238

IV. Epilogue **243**

8. Related Work 245
8.1. Approaches focused on Confidentiality 245
8.1.1. Access Control Models 246
8.1.2. Access Control Policy Analyses 248
8.1.3. Confidentiality Analyses 250
8.1.4. Usage Control Approaches 252
8.1.5. Industrial Tools & Approaches 253
8.2. Approaches focused on Attacks & Attackers 253
8.2.1. Vulnerability & Attack Classifications 253
8.2.2. Attack Path & Threat Modelling 255
8.2.3. Attack Path Estimation & Automatic Analysis 256
8.2.4. Industrial Tools & Approaches 261
8.3. Related Work Summary 262

9. Conclusion 265
9.1. Summary 265
9.2. Benefits 270
9.3. Future Work 272
Acknowledgement 277

V. Appendix **279**

Bibliography 281

A. Evaluation Results Effort Reduction Targeted Attack Graph Analysis 307

List of Figures

2.1. Metamodel layers illustration based on Stahl et al. [188] and MOF [2] . . . 19
2.2. Overview of the syntax for our metamodels . . . 19
2.3. Attribute Based Access Control (ABAC) access decision based on Hu et al. [73] with the naming schema from eXtensible Access Control Markup Language (XACML) . . . 23
2.4. A simplified ABAC architecture based on Hu et al. [73] . . . 24

3.1. Software architecture overview of the maintenance scenario based on [211] . . . 29

4.1. Metamodel elements to identify services for annotation . . . 40
4.2. Attribute metamodel with gray elements based on XACML . . . 46
4.3. Simplified access control policy metamodel with grey elements for elements based on XACML and white elements as new elements[211] . . . 50
4.4. Attribute provider and scenario metamodel for context-based policies . . . 54
4.5. Process for creating a XACML model and creating an access request 58
4.6. Excerpt PolicySet Transformation . . . 58
4.7. AllOf transformation . . . 59
4.8. EntityMatch, MethodMatch and GenericMatch transformation . . 61
4.9. SimpleAttributeSelection transformation . . . 63
4.10. Result model for PDP decisions . . . 67
4.11. Category metamodel elements based on [211] . . . 72
4.12. Vulnerability metamodel elements based on [211] . . . 73
4.13. Vulnerability Palladio Component Model (PCM) integration . . . 80
4.14. Approach for automatic extraction of vulnerabilities based on [94] 83
4.15. Attack metamodel elements based on [211] . . . 84
4.16. Attacker propagation metamodel elements based on [211] . . . 86
4.17. Attacker propagation instance for the running example . . . 87

4.18. Attacker with filtered attack paths metamodel elements 88
4.19. Attacker instance with filters for the running example 91

5.1. Process for using the access usage in a new system. Icon Source: Font Awesome by Dave Gandy — `http://fontawesome.io` 96
5.2. UsageScenario for the running example with the technician accessing the log data of the machine during a failure state 97
5.3. MisusageScenario for the running example with the technician accessing the log data of the machine 98
5.4. Result metamodel for the access usage analysis 104
5.5. Analysis result model for the running example 106
5.6. Process for using the attack propagation in a new system. Icon Source: Font Awesome by Dave Gandy — `http://fontawesome.io` 108
5.7. Attack propagation analysis steps 110
5.8. Resulting attack propagation graph for the running example with the starting point `Terminal` 125
5.9. Result and seed metamodel elements for the attack propagation analysis 126
5.10. Attack propagation result example for the running example in *Eclipse* 128
5.11. Process for analysing attack graphs 130
5.12. Targeted Attack Graph analysis process steps 131
5.13. Resulting attack graph for the running example without filters . . 135
5.14. Resulting attack graph for the running example with the vulnerability complexity filter for `Low` 137
5.15. Attack graph of the running example with identified paths from `Terminal` to the target element `ProductStorage` 138
5.16. Metamodel elements for the result of the targeted attack graph analysis 142
5.17. Attack path result example for the running example in *Eclipse* . . 143

6.1. Architectural overview TravelPlanner scenario 150
6.2. Architectural overview *Power Grid* scenario 153
6.3. Architectural overview *Target* scenario 157
6.4. Hardware resources and networks in the cloud infrastructure scenario 160
6.5. Simplified overview ABAC banking scenario 162

7.1. Overview of the GQM plan for contribution C3 171

7.2. Overview of GQM plan for contribution C4.1 182
7.3. Chained `ResourceContainers` schematics for the scalability evaluation 193
7.4. Scalability results (**G4**) for increasing number of resource containers 205
7.5. Overview of GQM plan for contribution C4.2 212
7.6. Scalability results (**G7**) for increasing number of resource containers 225

8.1. Exemplary simplified attack tree based on Schneier [162] 255

9.1. Overview of the approach related to the security terms defined by ISO 27000 [77] 266

List of Tables

3.1. Access control policies and vulnerabilities for the running example 34

6.1. Characteristics of the evaluation scenarios 167

7.1. Evaluation results regarding the JC for **Q1.1** and **Q1.2** 176
7.2. Evaluation results for the attack propagation analysis regarding accuracy and effort reduction 196
7.3. Activities performed during the usage of the attack propagation analysis 198
7.4. Evaluation results for the targeted attack graph analysis regarding accuracy and effort reduction 222

A.1. Evaluation results for **Q6** 307
A.1. Continued evaluation results for **Q6** 308
A.1. Continued evaluation results for **Q6** 309

List of Acronyms

PCM Palladio Component Model

ADL Architecture Description Language

APT Advanced Persistent Threats

SEFF Service Effect Specification

EMF Eclipse Modelling Framework

XACML eXtensible Access Control Markup Language

CWE Common Weakness Enumeration

CVE Common Vulnerabilities and Exposure

CVSS Common Vulnerability Scoring System

CWSS Common Weakness Scoring System

NVD National Vulnerability Database

ABAC Attribute Based Access Control

RBAC Role-Based Access Control

IoT Internet of Things

PDP Policy Decision Point

OASIS Organization for the Advancement of Structured Information Standards

XML Extensible Markup Language

UML Unified Modeling Language

VM Virtual Machine

KAMP Karlsruhe Architectural Maintainability Prediction

ICS Industrial Control Systems

OWASP Open Web Application Security Project

DSL Domain Specific Language

POS Point of Sale

GQM Goal Question Metric

CWSS Common Weakness Scoring System

CAPEC Common Attack Pattern Enumerations and Classifications

CPS Cyber-Physical System

Part I.

Prologue

1. Introduction

This thesis presents our approach for analysing the security of a given software architecture. In our security analyses, we consider vulnerabilities and access control policies. We explain why these are essential for security analyses in Section 1.1. In addition, we motivate why it is essential for security experts and software architects to consider the outcomes of our analyses. Section 1.2 outlines the problems we address in this thesis. These problems lead to our research questions, which we present in Section 1.3. Our contributions, which answer the research questions, are presented in Section 1.4, and this chapter concludes with an outline for the thesis in Section 1.5.

1.1. Motivation

Software systems have become ubiquitous in our daily lives, based on the emergence of innovative technologies, such as Internet of Things (IoT) and cloud computing. These advancements are being leveraged across various domains, including the manufacturing industry, where Industry 4.0 [161] and Industrial Internet of Things (IIoT) are gaining traction, the energy sector, with the rise of smart grids or the healthcare sector, through the implementation of smart health services. Among these types of system the utilization of technologies like IoT or Cyber-Physical System (CPS) for connecting numerous devices with software services is common. This results in a complex network of heterogeneous devices and services. These systems often handle sensitive data. Examples of such sensitive data can be personal health data in the case of smart health services or detailed production data in the case of Industry 4.0. Because of the sensitive nature of the data, the data should not be accessible to everyone. Therefore, these connected systems must handle the data confidential without providing access to external third parties.

Confidentiality is defined by ISO 27000 as part of information security [77, section 3.28]. Confidentiality ensures "that information is not made available

or disclosed to unauthorised individuals, entities, or processes" [77, section 3.10]. Access control, among other mechanisms, is a common means for achieving confidentiality. *Access control* means "to ensure that access to assets is authorised and restricted based on business and security requirement[s]" [77, section 3.1]. The access control system is specified by access control policies. These policies specify under which condition access to assets is granted. An asset is the entity, which needs protection. For instance, a service can be an asset. We call the access requesting entity in our work requestor. Typically, access is determined based on attributes of the requestor, such as their role. However, the specification of access control policies is very cumbersome, and the manual specification can be complex [204]. Because access control policies increasingly consider the context for the access decision, the policies themselves are getting more complex in contrast to non-context-based policies. The consideration of context attributes in the access control policies results in the access decision about accessing an element being context-dependent. In addition, the consideration of the context leads to more fine-grained access control policies than previous classical access control models, such as Role-Based Access Control (RBAC) [55]. Examples of such contexts can be the state of a machine in a production setting or the location of a user. Due to the complexity of context-based policies, the impact is often unknown. This increases the possibility of misconfiguration or misspecification. Misspecifications of policies are faulty access control policies. These policies grant access under the wrong conditions, such as too wide (permissive) or too restrictive policies. An example of too-wide access control policies is granting users access to services that should require root privileges. Misspecification of access control policies is problematic because they might block legitimate access requests [23] or enables misuse through malicious users. Both cases can be very costly. For instance, blocking legitimate access may prohibit users from performing their tasks. Therefore, it leads to delays in certain tasks or unusable systems. For instance, this might be blocking business processes within a business system. In a production process, this could mean a downtime of the assembly line, which is very costly[1][2]. On the

[1] S. Ravande. *Council Post: Unplanned Downtime Costs More Than You Think*. Forbes. Section: Innovation. Feb. 22, 2022. URL: `https://www.forbes.com/sites/forbestechcouncil/2022/02/22/unplanned-downtime-costs-more-than-you-think/` (visited on 06/06/2023).

[2] *Downtime Costs Auto Industry $22k/Minute - Survey*. Mar. 29, 2006. URL: `https://web.archive.org/web/20230420005357/https://news.thomasnet.com/companystory/downtime-costs-auto-industry-22k-minute-survey-481017/` (visited on 06/06/2023).

other hand, misuse by malicious users can lead to fines for confidentiality violations leading to privacy violations. For instance, H&M was fined 35.3 million euros for privacy violations[3], Amazon paid 5.8 million US Dollars for privacy violations[4], or British Airways was fined 20 million pounds[5]. It is also important to see that the mentioned fines did not take place in only one country but were imposed by authorities in different countries such as Germany or the USA. This shows the global importance of this issue. Besides the privacy implication also other implication can happen such as reputation loss [54] or leakage of confidential data. Therefore, it is necessary to analyse different scenarios with the context to estimate the impact of access control to avoid confidentiality violations. In addition, because systems evolve over time, not only the initial specification is important but also the changed system and specification need to be considered. It is also necessary to analyse the access control policies in an evolution step to estimate the impact of a policy change.

Besides the complex access control policies resulting in unknown confidentiality impacts, there are vulnerabilities in the used devices or products. A *vulnerability* is a “weakness of an asset [...] that can be exploited [...]” [77, section 3.77]. In our case, we concentrate on software vulnerabilities, such as the Log4Shell vulnerability [130], which enabled attackers to execute arbitrary code. Through the exploitation of vulnerabilities, attackers can gain access to previously restricted elements and data. This exploitation and gaining access can be a potential confidentiality violation. They need to be considered for a confidentiality analysis. Otherwise, the impact of vulnerabilities on confidentiality is unknown. The vulnerabilities are also directly linked to access control mechanisms because certain vulnerabilities require special privileges for exploitation, such as CVE-2009-07834 [125]. Other vulnerabilities may leak the credentials or authorisation, which enables attackers to circumvent

[3] The Hamburg Commissioner for Data Protection and Freedom of Information. *35.3 Million Euro Fine for Data Protection Violations in H&M’s Service Center*. Oct. 2020. URL: https://datenschutz-hamburg.de/assets/pdf/2020-10-01-press-release-h+m-fine.pdf (visited on 04/04/2023).

[4] D. Bartz. *Amazon’s Ring used to spy on customers, FTC says in privacy settlement*. June 1, 2023. URL: https://www.reuters.com/legal/us-ftc-sues-amazoncoms-ring-2023-05-31/ (visited on 06/05/2023).

[5] H. Beverley-Smith et al. *British Airways Faces Significantly Reduced £20M Fine for GDPR Breach*. The National Law Review. Oct. 23, 2020. URL: https://www.natlawreview.com/article/british-airways-faces-significantly-reduced-20m-fine-gdpr-breach (visited on 06/07/2023).

access control mechanisms, such as in CVE-2021-28374 [129]. Various studies show that vulnerabilities exist in systems, and it cannot be assumed that they will always be addressed once they are known. For instance, an HP study states that 70% of IoT devices are vulnerable[6]. A study by Unit 42, a threat intelligence team, shows high usage of vulnerable components in cloud providers[7]. Similar results can be found in a UK government study, stating that 32% of large businesses have outdated Windows installations [84]. A newer version states that in the production field, it is around 26% [47]. Outdated software often contains vulnerabilities. The importance of this issue is stressed even more by the fact that vulnerable and outdated components constitute their own category in the Open Web Application Security Project (OWASP) 10 [137]. The OWASP 10 [137] is a list compiled by OWASP which contains, in their opinion, the ten most critical security risks for web applications. Based on the fact that most modern applications have some kind of web interaction or use similar technologies, their findings are also transferable to other domains. Furthermore, the number of available vulnerabilities is increasing every year, while the average time for finding an exploit is decreasing [56, 153]. Addressing all these vulnerabilities is not always possible or feasible due to factors, such as costs or practicality [151]. For instance, in the production domain, an outdated machine controller may require a costly replacement of the complete machine. Furthermore, a report by IBM states that only 8% of known exploits are new[8].

Attackers often exploit an initial attack point to get access to a system and then propagate further in the system [84]. This propagation is often called lateral movement. This movement involves discovering new access privileges to further compromise the system [152, p. 65]. Others refer to this movement and exploitation of multiple vulnerabilities as vulnerability chaining, as it

[6] *HP Study Reveals 70 Percent of Internet of Things Devices Vulnerable to Attack.* July 2014. URL: https://web.archive.org/web/20190420151125/https://www8.hp.com/us/en/hp-news/press-release.html?id=1744676#.XLs2pBXP3ao (visited on 04/03/2023).

[7] J. Greig. *96% of third-party container applications deployed in cloud infrastructure contain known vulnerabilities: Unit 42.* en. Oct. 2021. URL: https://www.zdnet.com/article/96-of-third-party-container-applications-deployed-in-cloud-infrastructure-contain-known-vulnerabilities-unit-42/ (visited on 04/03/2023).

[8] *Website: IBM Security X-Force Threat Intelligence Index 2023.* en-us. Mar. 2023. URL: https://web.archive.org/web/20230318123413/https://www.ibm.com/reports/threat-intelligence (visited on 04/04/2023).

involves the sequential exploitation of multiple vulnerabilities[9]. Therefore, to obtain a comprehensive understanding of possible attack paths in the system, we need to consider both access control properties and multiple vulnerabilities. Understanding these attack paths is important for devising effective mitigation strategies. For example, a firewall can block access to a vulnerable component, thereby reducing the risk of exploiting the component's vulnerability. Consequently, considering only a single vulnerability or access control policy without the context cannot fully determine the impact on security.

Examining a software system for security properties is possible on different abstraction layers. For instance, various security analyses exist based on the network layers, such as Aksu et al. [8] or Yuan et al. [223]. However, they often fail to consider fine-grained access control properties or fail to distinguish between application and network layers. Commercial options, such as Bloodhound[10] or Codeshield[11], consider more fine-grained access control properties. However, they are limited to specific application domains, such as Microsoft's Active Directory or Amazon Web Services (AWS). Meanwhile, the security can also be analysed based on the existing source code. Different source code analyses have been developed [183]. These approaches commonly overlook security properties pertaining to the infrastructure or are restricted to a single programming language. Furthermore, the high level of detail in the source code often makes identifying overall design issues challenging.

In contrast to the mentioned approaches, a software architecture model can provide valuable information about the deployment and consider the infrastructure and application layers. Additionally, it can offer insights into the design of the software due to its higher level of abstraction. Considering the design is particularly important as security vulnerabilities often stem from design flaws. This notion is supported by OWASP's introduction of the element *Insecure Design* in 2021, which describes that the security flaw is already contained in the original design [1]. This also aligns with the observations made by McGraw [110]. In both cases, the basic idea is that the security can depend on the design. For instance, the parsing of a raw string without cleaning can be a security vulnerability. In this case, the design fault

[9] First. *Common Vulnerability Scoring System version 3.1: User Guide*. URL: `https://www.first.org/cvss/user-guide#3-4-Vulnerability-Chaining` (visited on 04/28/2023).
[10] *BloodHound Enterprise*. URL: `https://bloodhoundenterprise.io/` (visited on 03/20/2023).
[11] CodeShield GmbH. *Codeshield*. URL: `https://codeshield.io/` (visited on 03/20/2023).

is not filtering the string before parsing the string. These design decisions can also be found in a software architecture. For example, vulnerabilities or known security issues may be found in off-the-shelf components (bought components), and software architects could be aware of them. Consequently, by choosing a component, the software architect also chooses parts of the vulnerabilities for a system. Hence, the architectural design decision and, thereby, the software architecture affect the security properties.

Furthermore, analysing the system during the design phase is beneficial as rectifying flaws in later stages typically incurs higher costs [174]. Thus, considering the software architecture is beneficial because it allows to analyse the system during the design phase without requiring the system to be operational. A modelled software architecture remains useful even in latter phases because the software architecture enables analysis during downtime (e.g., maintenance tasks or attacks). In addition, a modelled system enables manually creating different what-if cases and analysing these to find the optimal solution. Therefore, analysing security properties based on the software architecture is beneficial.

This thesis aims to develop an approach that facilitates the analysis of access control policies and vulnerabilities using a software architecture model, thereby enabling the more secure design of software. In our scenario, the software architecture models are specified by *Software Architects*, similar like in PCM [154]. The access control properties and vulnerabilities are specified by *Security Experts* because of the required knowledge which goes beyond the usual knowledge of software architects. In the latter sections, we will specify the activities in more detail. The approach has the capability to identify potential access violations by both regular users and malicious actors. This approach gives a first impact on how access control policies affect the access decision of a system. Besides focusing on access control policies, the thesis investigates potential attack propagations by considering the exploitation of access control policies and vulnerabilities. The analyses provide security experts or software architects with possible attack paths. This gives an initial impact on vulnerabilities and whether they can be problematic or not. In addition, the attack paths give a first starting point for finding mitigation locations.

1.2. Problem Statement

Our motivation provides initial insights into the problems that we aim to address in this thesis. In general, our goal is to provide architectural security analyses for access control policies and vulnerabilities that aid in the design of secure systems. In other words, our objective is to assist security experts and software architects in constructing more secure software systems. In this regard, we identified two specific problems which we intend to tackle within this thesis.

P1: Unknown Security Impact of Access Control Policies The first problem we identified is the unknown impact of access control policies. Due to the complexity of context-considering access control policies, the impact is not always known. Especially in more complex systems like in Industry 4.0 with multiple different access control policies. This unknown impact manifests in two ways. The first is in the initial design or specification of the policy, and the second is during an evolution change. For a single access control policy, this might not be problematic. For instance, an exemplary access control policy in an Industry 4.0 setting might be that access is granted only during a maintenance task. This single access control policy is very simple, and the effect is relatively easily understood. However, if this policy is integrated with a system, the protected system entity interacts with other entities, which are also protected by other policies. Hence, the successful interaction depends on multiple different access control policies. For instance, it could be that for a maintenance task, also other services are needed, which are protected by different access control policies. Therefore, additional access control policies need to be considered to evaluate whether the actual maintenance task can be fulfilled. In addition, more and more access control policies consider the current context for an access control decision. This further complicates the access decision because the access decision depends on the context. Therefore, the impact can only be estimated by considering the context.

Additionally, access control policies can change over time, in order to accommodate new processes or tasks. In this scenario, a similar issue may arise. The problem lies not only in the uncertainty surrounding whether a particular task or process can be executed but also in the potential widening of access definitions resulting from policy changes. As a consequence, these changes may inadvertently grant access to malicious users.

Ultimately, this problem is a trade-off decision between achieving the system's goal and ensuring security. In addition, it can be viewed as a cost issue, as accidentally blocking legitimate access requests may block or hinder a business process. Therefore, the users need to invest additional time to complete their tasks. Also, too permissive (too-wide) access control policies can be considered a cost problem because they can result in loss of knowledge resulting in losing a competitive edge. Furthermore, in some cases, it also results in fines. To avoid this problem, it is necessary to analyse access control policies regarding the used scenario and in combination with the system.

P2: Unknown Security Impact of Vulnerabilities The second problem we identified is the unknown impact of vulnerabilities. As mentioned earlier in the motivation, the number of vulnerabilities continues to increase. This leads to the challenge for security experts to identify which vulnerabilities should be addressed first. Even more, in some cases, it is not feasible to address them at all due to limited resources or the absence of available patches. Therefore, it is necessary to manage the vulnerabilities and identify the critical vulnerabilities [151]. To accomplish this, understanding the impact of a vulnerability is important. However, similar to P1, the impact of a vulnerability cannot be completely determined independent of the used system. For instance, a vulnerable component can enable an attacker to reach other components, such as a gateway component. However, in other cases, there might be no connected components. In other scenarios, an attacker cannot exploit a vulnerability because it is only locally exploitable, and the attacker has only access over a network. Furthermore, certain vulnerabilities require specific permissions for exploitation, or they grant certain permissions that can be used later for further propagation. Therefore, it is unclear what a vulnerability's actual impact is and whether it enables further attack propagation.

This problem can also be viewed as a cost problem. For instance, in the case of vulnerabilities affecting confidentiality, the same cost problem regarding leaked data as for P1 arises. Another problem is the vulnerability management. For instance, one solution for addressing the unknown impact is for security experts to secure all vulnerabilities automatically. However, the cost could be very high because of the vast number of vulnerabilities. In addition, in scenarios involving production controllers, the replacement of an entire machine may be required, further increasing the costs. For solving this problem, we need attack analyses, which can assess the vulnerability.

One possible solution for assessing a vulnerability is detecting whether the vulnerability is used in an attack path. Based on the assessment results, the software architects can discuss with the security expert whether the identified attack path needs mitigation. If mitigation is deemed necessary, the security expert can propose different mitigation techniques to the software architects to break the attack path towards the critical components.

1.3. Research Questions

Based on the problem description, we derived our research questions. These research questions help in narrowing down our research and guided us in the development of our approach. We defined the following research question:

RQ1 How can violations of access control policies in context-dependent scenarios be identified in relation to the software architecture?

- **RQ1.1** How can access control policies for context-dependent scenarios be modelled in relation to the software architecture?
- **RQ1.2** How can access control policies for context-dependent scenarios be analysed in relation to the software architecture?

RQ2 How can we identify attack propagations based on the software architecture?

- **RQ2.1** What are relevant architectural properties for an attack propagation?
- **RQ2.2** How can we analyse a software architecture for attack propagations?

The first research question, as outlined in **RQ1**, investigates how violations of access control policies can be identified. This is especially relevant to context-based scenarios as commonly found in IoT-enabled environments, such as those encountered in Industry 4.0. Moreover, the identification should be possible based on the software architecture and thereby enabling the analysis during the design time. This is similar to our definition of dynamic changes, which states that dynamic events need to be foreseeable at design time [215]. Dynamic events are usually also context-dependent, such as the position of a worker. We transfer this definition to our access control policies, meaning that

we only investigate access control policies, which can be expressed during design time. This definition excludes some policies, such as self-adapting policies, but it simplifies the approach itself. The first sub-research question **RQ1.1** narrows our research to software architecture and modelling access control policies. This is necessary because before we develop an access control analysis, we need to specify the input for the analyses. In our case, the input is the software architecture and the access control policies. Therefore, we must first investigate how access control policies can be modelled. The second part is the integration of access control policies into software architecture models. This is necessary because access control policies usually protect assets. In our case, we described these assets as elements within a software architecture. Therefore, we need an integration for access control policies within the software architecture. However, the access control policies should depend on the software architecture, but not the other way round. In other words, a change in the access control policies should not require a change in the software architecture. This means the access control policy model should be separate from the software architecture model and only reference the software architecture model. The second sub-research question **RQ1.2** then investigates the actual analysis of access control policies. The analysis is dependent on the software architecture model, meaning that the analysis uses the software architecture to analyse the access control policies.

The second research question **RQ2** is derived from problem P2. The questions consist of two essential parts. First, our focus is on attack propagation. Secondly, we base our analysis on the software architecture. Similar to our first research question, the sub-research question **RQ2.1** investigates the relevant properties for an attack propagation. Identifying the relevant properties is necessary because the properties are the foundation for our modelling language to specify the input of the attack analysis. Our goal here is not to provide a complete list of properties but to identify the most critical aspects. Once again, we utilize the software architecture to describe the assets. In this problem, the assets are vulnerable architectural elements. The second sub-research question **RQ2.2** then targets the analysis part. It specifies the desired outcome of an attack analysis. In addition, it investigates how the results can be used.

The research questions are answered by our contributions, which we present in the next section.

1.4. Contributions

Based on our research questions, we derived our four main contributions. These consist of different modelling languages and analyses. We will first outline each contribution and then provide a more detailed description.

Contribution C1: Access Control Modelling Language for Software Architectures The first contribution C1 addresses the research question **RQ1.1**. It is a modelling language which enables software architects or security experts to define access control policies for different architectural elements, such as components, services or hardware devices. In addition, it provides support for modelling context-dependent scenarios. As the access control of a system is crucial for understanding attack propagations, this contribution also aids in answering research question **RQ2.1**. Our access control modelling language is built upon the existing standard for ABAC policies. Leveraging this foundation allows us to define context-aware access control policies, which are essential in IoT scenarios. In addition, using an existing standard can ease our approach's application because experts can already be familiar with the concepts. The modelling language provides the explicit representation of access control policies for various software architectural elements, thereby enhancing the documentation of security properties. Moreover, the modelled policies can serve as a foundation for effective communication among diverse stakeholders and experts. The explicit modelling of access control policies also enables the design and analysis of such policies. The later analysis can lead to the design of more secure policies because the analysis can identify misspecified policies. Consequently, this contribution can help to design more secure systems.

Contribution C2: Vulnerability Modelling Language for Software Architectures The second contribution C2 is our vulnerability modelling language featuring vulnerabilities, attacks and attackers. Together with C1, it answers the research question **RQ2.1**. The modelling language is built upon existing industrial vulnerability classifications. It can assign vulnerabilities to different architectural elements, such as components or hardware devices. Furthermore, it models additional information concerning the exploitability of a vulnerability. For instance, it defines the attack vector, which describes from which location the vulnerability can be exploited. Additionally, the modelling

language encompasses elements for modelling attackers and attacks, which is essential for distinguishing between different attack types. Different attacker types are necessary as vulnerabilities exhibit varying different properties for exploitations. For instance, the level of complexity may vary for attacks. Therefore, some attacks may require more sophisticated attacker capabilities than others. Our vulnerability modelling language provides means for security experts to model relevant properties of vulnerabilities and annotate them onto a software architecture. This modelling serves to document vulnerabilities and forms the foundation for subsequent architectural security analysis. Ultimately, by making vulnerabilities explicit and analysable, this modelling language can help to improve system security.

Contribution C3: Scenario-based Access Usage Analysis The third contribution C3 is a scenario-based access usage analysis. By utilising our contributions C1, this contribution enables analysing access control policies through usage scenarios. It answers our research question **RQ1.2**. Software architects can specify different usage scenarios and enhance the scenarios with context information. Each scenario contains the initial system calls by the users and is then analysed to determine whether the system calls are feasible based on the current context and access control policies. This analysis is especially beneficial for software architects and security experts looking to investigate and understand how access control policies function within specific contexts and scenarios. In the event of a violation, software architects can adapt the scenarios, or security experts can modify the access control policies to mitigate the violation. Furthermore, the analysis supports the analysis of malicious scenarios. These are scenarios which should not be possible given the provided access control policies and contexts. This concept is useful for analysing whether forbidden behaviour is achievable or not. This contribution enables the security expert to identify excessively permissive or overly restrictive access control policies based on the intended usage. Hence, they can design more secure access control policies by determining the appropriate level of permissiveness required. Furthermore, software architects can ascertain whether the intended usage can be achieved through the specified access control policies.

Contribution C4: Architecture-based Attack Analyses Our last contribution C4 is the attack analyses. This contribution consists of two sub-contributions.

Both contributions use the modelling languages from the contributions C1 and C2 and answer the research question **RQ2.2**.

Contribution C4.1: Architecture-based Attack Propagation Analysis The first attack analysis C4.1 examines a software architecture for potential attack propagation. A similar scenario can be observed in real systems with insider attacks, where an insider exploits the system to get more privileges. The analysis requires a starting point and a dedicated attacker model. The analysis subsequently provides a list of affected elements. These affected elements are the elements an attacker can reach from the start point and is able to compromise. Hence, this analysis is important, for instance, to identify how far an attack can spread within a system. Our propagation uses credentials together with vulnerabilities and calculates the affected elements based on the software architecture.

Contribution C4.2: Architecture-based Targeted Attack Graph Analysis The second analysis C4.2 is a targeted attack analysis. In contrast to C4.1, it does not require a starting point but requires a specified target. The analysis calculates attack paths leading to the target. This analysis is beneficial when software architects or security experts want to identify whether there exist an attack path to the targeted element. This is important, for instance, if the targeted element is a critical component, such as a banking component. In such cases, identifying an attack path to the component signifies a risk, as it indicates that attackers can potentially compromise this critical component. It is, therefore, important to be aware of this risk and consider potential mitigation strategies.

The results of both attack analyses (C4.1, C4.2) can be used by security experts in collaboration with software architects to evaluate the security of a system. Both analyses provide a list of affected elements, which indicate potential compromised elements by an attacker. Based on these affected elements, security experts can then identify appropriate mitigation actions, such as implementing a firewall or modifying the security configurations, like the access control policies and discuss the integration of the mitigation with the software architects. Implementing the identified mitigations can disrupt attack paths, thereby enhancing the system's overall security.

1.5. Outline

The remainder of the thesis is structured as follows. In Chapter 2, we introduce the foundations of our approach. The foundation covers the introduction of PCM as our chosen Architecture Description Language (ADL) and the introduction of security approaches and terms. We introduce in Chapter 3 our running examples, which we use to explain the core concepts in our approach. Afterwards, we describe in detail our contributions in Part II. We start by describing our developed modelling languages in Chapter 4. This description contains the modelling languages for access control properties, vulnerabilities, attacks, and attackers. These modelling languages are the foundation for our analyses introduced in Chapter 5. After describing our contributions, we describe our validation in Part III. We start by describing our evaluation scenarios in Chapter 6. These consist of different scenarios based on real-world scenarios and research scenarios. We use them in our evaluation described in Chapter 7. There, we describe our evaluation goals and give the results for each developed analysis. Our thesis concludes with the Part IV. It covers the related work in Chapter 8 and in Chapter 9 the conclusion with a summary of the thesis and an outlook for potential future work.

2. Foundations

In this section, we explain the foundations upon which our thesis is built. Our approach relies on the modelling and analysis of software architectures. In Section 2.1, we explain the key concepts relevant to our software architecture modelling. It explains modelling concepts in general and describes a concrete ADL. The second part of our foundations is the definition and introduction of security-related terms in Section 2.2. There, we define and introduce the most important security concepts for our approach. The last section (2.3) introduces a categorization of threats to validity, which is important for the evaluation.

2.1. Software Architecture Description and Analysis

In this section, we will first explain in Section 2.1.1 the concept of *Model-Driven Software Development*. We give a definition of a model and explain the relationship between metamodels and models. Afterwards, we introduce in Section 2.1.2, the ADL which we use in our approach to model the software architecture. We describe in the section the most important architectural modelling elements for our approach. Afterwards, we introduce in Section 2.1.3 the Karlsruhe Architectural Maintainability Prediction (KAMP) approach, which is a foundation for our attack propagation.

2.1.1. Model-Driven Software Development

Model-Driven Software Development (MDSD) is a concept for developing software based on abstract models [188]. Developers specify models that are automatically transformed into source code. One of the aims of this process is to increase the efficiency of software development [188, section 1.1]. Our approach also takes advantage of key concepts from MDSD.

Essential to the application of MDSD concepts is a common understanding and definition of a model. Stachowiak [187, p. 131ff] defines a model with the three properties *representation*, *abstraction* and *pragmatics*. The first property requires a model to be a representation of a specific entity. For example, a model of a vulnerability is a representation of the real vulnerability. The second property is that the model is an abstraction of the actual entity. In other words, the model is reduced. For example, a real software vulnerability may contain many aspects, such as the source code leading to the vulnerability. However, not all of this information is relevant. This is especially true in combination with the last property, the pragmatics. The pragmatics describe the purpose for which a model is created. For example, in our case, the model is created to analyse the propagation of an attack.

In our approach, we use models to describe a software architecture and its security properties. Based on the models, we provide automatic analyses that can determine different security qualities, such as access violations or accessible elements by an attacker. In order to be able to analyse the system automatically, we need to define a formal syntax of how these models should look like. Stahl et al. [188, p. 58ff] call models based on a structure *formal models*. This structure is called “a metamodel [which] describes the possible structure of models – in an abstract way, it defines the constructs of a modelling language and their relationships, as well as constraints and modelling rules [...]” [188, p. 85]. An example of such a metamodel is the Unified Modeling Language (UML) [3]. Furthermore, the UML metamodel is also described as a metamodel, which is then called a meta-metamodel. In our example case, this is the *Meta Object Facility (MOF)* specified by the Object Management Group (OMG) group [2]. This concept of layering between different structures can be repeated with an arbitrary number of layers. However, usually, four or fewer layers are sufficient [2]. In our approach, the meta-metamodel is the Ecore model from the Eclipse Modelling Framework (EMF) [191]. A similar layering to MOF and Stahl et al. [188] is shown in Figure 2.1. On the left side, the figure shows the different meta-layers and their relationship. On the right side, we give an example of our use case. As the meta-metamodel, we use *Ecore*. This provides us with concepts, such as classes. The metamodel defines our structure *Vulnerability*. The model defines the concrete vulnerability. In our case, this is the identification string *CVE-2021-28374*. M0 is then the concrete instance of the described vulnerability.

In our thesis, we used the concepts of metamodels to express the modelling languages of our contributions C1 and C2.

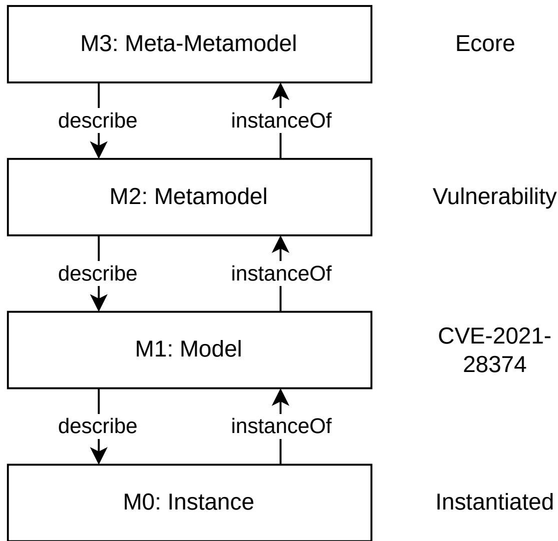


Figure 2.1.: Metamodel layers illustration based on Stahl et al. [188] and MOF [2]

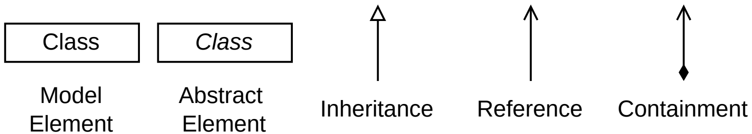


Figure 2.2.: Overview of the syntax for our metamodels

Figure 2.2 illustrates the graphical syntax we use in our illustrations to describe our metamodels. The syntax is based on the concept of UML class diagrams. Thus, rectangles with a name are a model element. In some cases, where the attributes are important, the rectangle is divided into the name section and an attribute section. Names in italic indicate an abstract element similar to an abstract class. Inheritance or generalisation is indicated by white closed arrows. Open arrows without a composition are references. Containment relationships are indicated by a composition (black-filled rhombus) and an open arrow. In cases where it is important, the multiplicity is given.

2.1.2. Software Architecture and Palladio Component Model

Our approach uses the software architecture as a foundation to describe a system. As a metamodel for the software architecture, we choose to use the Palladio Component Model (PCM) [154]. PCM is a well-established ADL. It supports the component-based development process and provides various quality analyses, such as performance or reliability [154]. Furthermore, there exist also different security analyses for PCM, such as Seifermann [166] or Pilipchuk [143]. Therefore, using PCM is beneficial because of its wide variety. This enables architects to reuse parts of the model and distribute the modellings costs across multiple quality analyses.

In this description, we will focus only on the elements of PCM that are relevant to this thesis. We begin by describing the *Repository*. The repository contains the components and interfaces. These are linked by required and provided roles. The interfaces contain the service declarations and are implemented in the components providing the interface. The service declaration is described by a `Signature`.

In the context of the PCM, the implementation of a service is referred to as Service Effect Specification (SEFF). SEFFs can invoke other services, which is denoted `ExternalCall` within PCM. In our case, a single component is represented by the model element `BasicComponent`. A `CompositeComponent` is a composition of several `BasicComponents`. The modelled components are instantiated in the *System* or *Assembly*. An instantiated component is referred to as an `AssemblyContext`. Different `AssemblyContexts` are connected by an `AssemblyConnector`. This connection is the actual wiring between the different provided and required roles of a component. In addition, the model contains the selection of the public interface for a system.

The deployment of the components is modelled in the *Allocation* containing the deployment relationship between the instantiated components and the hardware devices. The hardware devices are modelled in the *ResourceEnvironment*. It includes the hardware devices for processing nodes. These are called `ResourceContainers`. In addition, it contains network devices, which connect different `ResourceContainers`. The network devices are called `LinkingResources`.

The user behaviour is modelled in the *Usagemodel*. It groups the user behaviour into different `UsageScenarios`. These scenarios can be seen as usage

profiles [154, p. 103]. Within a `UsageScenario`, the different system calls to the public services specified in the assembly are modelled. This service call is referred to in PCM as `EntryLevelSystemCall`.

The previous description and the following only cover Palladio 5.1[1]. We do not include or consider incubation projects or other projects which are not part of the official PCM release.

2.1.3. Karlsruhe Architectural Maintainability Prediction Approach

The Karlsruhe Architectural Maintainability Prediction (KAMP) [189, 190, 156, 71, 155, 36] approach is a change impact analysis for maintenance tasks. Based on a given software architecture, it calculates potentially affected architectural elements for a given change request. The approach propagates the original change through the structural elements of a software architecture. It was successfully used in the context of software architectures [189, 156], business systems [155] and automation systems [71]. The basic idea is to define a propagation rule for each type of change in a software architecture. For instance, a propagation rule could be that a changed service name affects the implementing service. Here, the change type would be the name change and the propagation rule would be the relationship to the implemented service. This propagation can then continue because the implemented service is now affected. This could then propagate, for instance, to the services requiring the implementation. This propagation continues as the affected service then impacts other dependent services, and so on, until no new elements are affected. In the end, the analysis then provides a list of affected elements. This behavior bears similarity to attack propagation, where attackers also propagate through structural architectural elements. As with maintenance, this propagation follows certain propagation rules, such as in the case of attackers that they exploit vulnerabilities or knowledge of the system. Attackers also usually continue until they either achieve their goal or cannot further propagate to other elements. In the case of maintenance tasks, the analysis also propagates till the change does not affect any other architectural elements. In addition, the expected output is similar because, in both cases,

[1] *PCM 5.1 - SDQ Wiki*. URL: `https://sdq.kastel.kit.edu/wiki/PCM_5.1` (visited on 06/06/2023).

we expect the affected elements and a reason for it. Therefore, we decided to use this propagation structure as our foundation for our attack propagation and investigated whether it is useful.

2.2. Security Related Concepts and Terms

In this section we discuss important security terms and techniques. First, we explain our used access control concept in Section 2.2.1. Then we describe security incidents in Section 2.2.2 and our used vulnerability concept in Section 2.2.3. In the last section 2.2.4 we describe the concept of misuse cases to model malicious activities.

2.2.1. Access Control Specification

As described in the introduction, confidentiality is a property of information security and can be defined as "that information is not made available or disclosed to unauthorized individuals, entities, or processes" [77, section 3.10]. Other security properties for information security can be availability or integrity [77, section 3.28]. However, in our work, we focus on confidentiality. Access control is one technique that can be employed to ensure the non-availability or disclosure of information to unauthorized parties. Other approaches can be, for instance, information flow control [50] or encryption [22]. In our approach, we focus on access control to preserve confidentiality. During access control requests, users typically request resources and authenticate themselves to the system. This process includes the security property authenticity, which is "[...] that an entity is what it claims to be" [77, section 3.6]. In our work, we abstract from the authentication process itself. We consider entities to be authenticated if they have the appropriate attributes for accessing a resource. However, our approach does not consider how the attributes are assigned to entities.

During the introduction, we highlighted the importance of context-considering access control policies to provide access control policies for context-dependent scenarios. To address this, we choose Attribute Based Access Control (ABAC) [73] as our access control system. ABAC is considered a dynamic access control approach and can take context into account during an access control decision. The main concept of ABAC is that multiple attributes

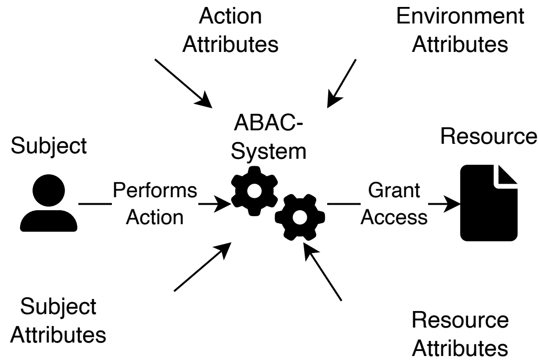


Figure 2.3.: ABAC access decision based on Hu et al. [73] with the naming schema from XACML

are evaluated during an access control decision. Figure 2.3 illustrates this behaviour. A subject (here user) wants to perform an action on a resource. An action represents an operation such as *read*. A resource can be any element upon which an action is performed, such as a file or a service. The ABAC system divides attributes into four distinct categories: subject attributes (e.g., user name or role), action attributes (e.g., name of the performed action), the resource attributes (e.g., name of the requested resource), and the environment attributes (e.g., time). All of these attributes are considered during the access control decision process. Afterwards, the access is either granted or denied.

A simplified architecture of an ABAC architecture is given in Figure 2.4. The figure is based on the description of the ABAC functional points in Hu et al. [73]. However, we excluded certain parts which are not relevant to this thesis. The `Requestor` is the requesting subject. It sends its request for access to a resource to the `Policy Enforcement Point` (PEP). The PEP "[e]nforces policy decisions in response to a request [...]" [73, p. 25]. The PEP then delegates the request to the `Policy Decision Point (PDP)`. The PDP performs the actual access control decision and returns the decision to the PEP. The PEP can then grant access to the requested resource handled by the `Resource Handler`. This separation between PEP and PDP is important for our analysis because it enables us to delegate the actual access decision process in our analysis to another component.

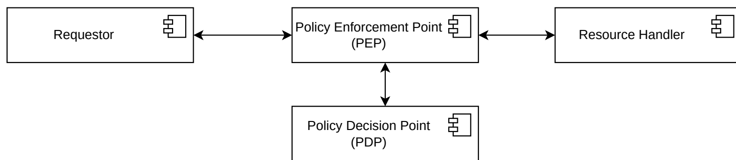


Figure 2.4.: A simplified ABAC architecture based on Hu et al. [73]

An implementation for ABAC is eXtensible Access Control Markup Language (XACML) [73, 132]. XACML is an open industry standard developed by the Organization for the Advancement of Structured Information Standards (OASIS). It provides a metamodel for access control and also provides XML schema files for the metamodel [134]. XACML defines a structure for access control and a mapping for the concrete selection of access control policies to resources and actions.

2.2.2. Security Incidents and Vulnerabilities

Our analyses provide feedback regarding access violations and potential attack paths, which can be regarded as potential *security incidents*. Security incidents consist of multiple unwanted *security events* which can affect a business process or system [77, section 3.31]. A security event is an "occurrence [...] indicating a possible breach [...]" [77, section 3.30] of elements or access control policies. In our analyses, these events are the successful attacks or the access violations. The attacks or the credentials of attackers represent potential *threats* to a system. A threat is defined by ISO as a "potential cause of an unwanted incident, which can result in harm to a system or organization" [77, section 3.74]. A threat also has a relationship to vulnerabilities because a *vulnerability* is a "weakness of an asset [...] that can be exploited by one or more threats" [77, section 3.77]. Based on this definition, for instance, a misconfigured access control policy is a vulnerability and an attacker exploiting this policy is a threat. Another example is the Log4Shell vulnerability [130]. The vulnerability is the implemented weakness in the software and the threat is the attacker exploiting it. In our thesis, this is an attack or an access violation. In the conclusion (c.f. Chapter 9), we will describe how our artefacts relate to the defined terms in more detail.

2.2.3. Vulnerability Classification

In this thesis, we define a vulnerability based on different existing vulnerability classifications. These classifications serve as a foundation for our attack propagation analysis. We can benefit from existing knowledge by leveraging the reuse of these established classifications. This reuse is beneficial because we can save time and use established techniques.

The first classification we describe is the Common Weakness Enumeration (CWE) by Mitre [111]. It classifies different weakness types, such as the usage of hardcoded credentials [116] or cleartext storage of sensitive information [114]. In addition, it groups the categories into parents and subgroups and thereby creating a hierarchy between different weakness types. However, these weaknesses are not specific vulnerabilities but describe general types of vulnerabilities.

Concrete software vulnerabilities are commonly identified and listed using a unique identifier known as Common Vulnerabilities and Exposure (CVE). For instance, CVE-2021-44228 [130] is one of the Log4Shell CVE vulnerabilities. These CVE IDs can often be found at vulnerability databases, such as the National Vulnerability Database (NVD) [131] or security advisor sites, such as for Microsoft[2] or for Ubuntu[3]. Usually, the databases also link a concrete vulnerability to its corresponding CWEs to provide more insights regarding the vulnerability.

In addition, these vulnerability databases typically provide a detailed textual description and assign a scoring based on the Common Vulnerability Scoring System (CVSS) [45]. CVSS is an approach to quantify the severity of vulnerabilities. It uses different metrics and metric groups to calculate the severity. While our specific use case does not require the severity score itself, the metrics used in CVSS are still relevant. Here, it is especially the *Base Metric Group*. This group describes the exploitation, such as the attack vector from which an attacker can exploit the vulnerability, or the impact, for instance, on confidentiality. Especially relevant is the paragraph that the CVSS specification "requires as a condition of use that any individual or entity which publishes scores conforms to the guidelines described in this

[2] *Security Update Guide*. URL: `https://msrc.microsoft.com/update-guide/vulnerability` (visited on 01/17/2023).

[3] *CVE reports*. URL: `https://ubuntu.com/security/cves` (visited on 01/17/2023).

document and provides both the score and the scoring vector so others can understand how the score was derived" [44]. In our case, this requirement is very beneficial because it requires that the individual metrics need to be published. Therefore, it allows us to reuse them later in our modelling.

2.2.4. Misuse Case

The concept of *misuse cases* was introduced by Sindre et al. [177]. Their basic idea is to model not only regular use cases, which represent the "good" behaviour of a user but also use cases that describe the behaviour of a malicious user, representing the "bad" behaviour. These use cases, describing malicious behaviour, are called misuse cases. Moreover, they establish a relationship between misuse cases and regular use cases using *mitigation* and *threaten* relationships. To implement this concept, they extended UML use cases to include misuse cases. Building on this idea, Sindre [176] proposed the *mal-activity diagrams.* The core idea is similar to that of misuse cases, but it is based on activity diagrams. The modelling elements in mal-activity diagrams use similar syntax and semantics to regular activity diagrams, with the difference that they describe malicious behaviour. To bring both approaches together, El-Attar [17] proposed an approach that combines the strengths of both misuse cases and mal-activity diagrams. Additionally, both approaches primarily target the requirement phase of the software development process.

2.3. Categorising Threats to Validity

During our evaluation, we categorise the threats to validity based on the validity categories in "*Guidelines for conducting and reporting case study research in software engineering*" by Runeson et al. [157]. In the following, we describe the different validity categories in general and then explain in the evaluation section (c.f. Chapter 7) the application to our evaluation.

- *Internal Validity:* This category considers the relationship between the observed effects and the reasons for the effects. During a study, researchers identify factors influencing the study results. However, the identified factor can also be influenced by a third factor. If this third factor is neglected or unknown, "[...] there is a threat to the internal validity" [157, p. 154].

- *External Validity:* This category investigates the generalisability and usefulness for other researchers. When investigating a case or scenario, the findings might be very specific to the studied case. The results might not be transferable. Hence, it is important to identify "[...] to what extent the findings are of relevance for other cases" [157, p. 154]. The idea is to find and use case studies (here: scenarios) that have "common characteristics" [157, p. 154] and allow researchers to generalise their findings.

- *Construction Validity:* This category describes, whether the "operational measures" [157, p. 154] align with the researcher's intention. In other words, whether the used techniques help to answer the research questions or, in our case, the evaluation questions. For example, it discusses the appropriateness of the evaluation metrics to the defined evaluation goals and questions. Hence, it can be described as whether the defined construction allows deriving the suggested conclusions.

- *Reliability:* This category describes how relevant the specific research is for creating the findings. In other words, " [...] if another researcher later on conducted the same study [or scenario], the results should be the same" [157]. There are different means to achieve this. One way can be a reproduction package, which allows other researchers to verify more easily. The importance of these packages is also emphasised by the introduction of ACM badges for *Artefact Reviews* [4]. Konersmann et al. [97] state also the need for replication packages (in the sense of reproduction) in software architecture research.

3. Running Example

Our running example is based on a scenario from Al-Ali et al. [12]. It is set in an Industry 4.0 environment and was developed within Trust 4.0[1], which was a joint research project with industrial partners. We extended the basic scenario in Walter et al. [211] by adding a software architecture with access control policies and vulnerabilities. In the following, we will describe the scenario and the software architecture in more detail.

The scenario consists of two companies: Producer P and Service Company S. P produces various goods by using a machine. During production, the machine generates log data. The log data is classified as confidential due to its potential inclusion of sensitive information, such as employee details or detailed

[1] *Trust 4.0 Dataflow-based privacy for industry 4.0.* Jan. 19, 2019. URL: `https://web.archive.org/web/20210422205559/http://trust40.ipd.kit.edu/home/` (visited on 06/06/2023).

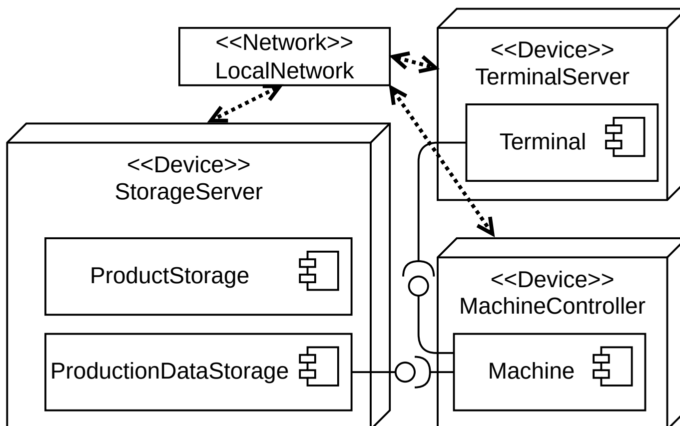


Figure 3.1.: Software architecture overview of the maintenance scenario based on [211]

information about the production process. Because of this confidentiality classification, S, which provides a service technician to maintain or repair the machine of P, is not granted access to the log data during normal production. However, in the event of an incident, such as a breakdown of the machine, a service technician from S can access the data from the machine. This allows them to resolve the incident so that the production can continue. There are two interesting aspects to this scenario. First, it includes access restrictions based on the changing state of the machine. This means that access is only granted during the incident. Therefore, it requires access control policies that support dynamically changing context attributes. Second, an external entity (here, the service technician) is accessing confidential data. Even though this access is only granted during a special situation, it allows attackers to exploit this situation to attack the system.

The software architecture is illustrated in the Figure 3.1. For illustration purposes, we decided against the separate UML notion of deployment and structural view. In our case, we combine the deployment and structural view. In the following, we will explain the syntax and semantics of the figure in more detail. We will use the same notion for the other examples. Figure 3.1 shows the components represented by the rectangles with the component symbols and their connection to other components. The logical connections between the components are illustrated by the provided and required connectors. The components are deployed on hardware resources. The hardware resources are marked with the stereotype «`Device`» and the deployment of the components is represented through the containment within the hardware resource box. Network elements are illustrated by the stereotype «`Network`». The dashed lines with arrow heads at both end between devices and network devices represent a network connection. This means that two devices which are connected to the same network device can communicate with each other.

The service technician uses the `Terminal` to access the log data. The component is deployed on the `TerminalServer`. The `Terminal` is connected to the `Machine` which is deployed on the `MachineController`. The log data is stored on the `ProductionDataStorage`, which is deployed on the `StorageServer`. The `StorageServer` is a dedicated server for storing data of P. Hence, the server runs also other storage components, such as the `ProductStorage`. The `ProductStorage` is a storage component, which is not considered in the initial scenario. Therefore, it has no direct connection to the other components. In our case, it symbolises a component that contains very sensitive data, such as the blueprints of P's products. Losing this sensitive data would be

considered a worst-case scenario in our example as P's competitors could then use the blueprints for their products. Each hardware device is connected by the `LocalNetwork`.

The attributes required for the access control are listed in Table 3.1. It shows for each architectural element the attributes required to access it. The attribute conditions are a conjunction and a requestor needs each attribute so that access can be granted. The requestor is the entity requesting access to the architectural element. Additionally, the table lists the vulnerabilities in this scenario. The access control policies in this scenario specify only a minimal set and not all elements have a policy. In the absence of a policy, access should be denied by default. The vulnerability in our running example is CVE-2021-28374 [129]. This vulnerability can leak in certain situations the credentials. In our case, we assume that it will leak the `Admin` attribute.

Remark: In this work, we assume that CVE-2021-28374 still belongs to the category CWE-312 [114]. During the writing of this thesis, the vulnerability was remapped from CWE-312 to CWE-732 [115]. This change is documented in the change log of the corresponding NVD website [129]. However, this assumption does not affect the underlying approach.

Based on the description of our running example, the following interesting tasks regarding the confidentiality can be derived. First, software architects want to determine whether the system is capable of executing the described scenario with the specified access control policies without any access violations. It is important that no access violations occur because otherwise, the violations can prevent users from performing their intended tasks. This can result in high costs. For example, if a service technician is denied access to a broken machine, the machine cannot be repaired, resulting in production stoppages. As previously described, these unplanned production stops are very expensive[2,3]. It is therefore important that the modelled software architecture and access control policies can be analysed for violations to prevent false access violations.

[2] S. Ravande. *Council Post: Unplanned Downtime Costs More Than You Think*. Forbes. Section: Innovation. Feb. 22, 2022. URL: `https://www.forbes.com/sites/forbestechcouncil/2022/02/22/unplanned-downtime-costs-more-than-you-think/` (visited on 06/06/2023).

[3] *Downtime Costs Auto Industry $22k/Minute - Survey*. Mar. 29, 2006. URL: `https://web.archive.org/web/20230420005357/https://news.thomasnet.com/companystory/downtime-costs-auto-industry-22k-minute-survey-481017/` (visited on 06/06/2023).

Second, software architects are interested in whether the access control policies can prevent malicious usages. For instance, a malicious usage, in our case, is the access of the external technician without the machine being in a failure state. In this case, the external service technician has no legitimate reason to access the machine and therefore, the access control policies should prevent the access. Hence, it is important that these malicious usages can be analysed to determine whether the access control policies prevent them.

Third, the software architect wants to determine how secure the architecture is. One possibility to estimate the security is to identify how far an attack attacker can propagate within the system. This propagation is essential to determine the potential threat to the system from an attacker. For instance, in our running example, the external service technician could be a malicious user and attack the system. In this case, it is important to identify which architectural elements can be reached starting from the `Terminal`, which is accessible by the technician. Based on the list of affected elements, the security experts can work together with the software architect to integrate appropriate mitigation strategies, such as changing access control policies or replacing components with more secure ones. It is, therefore, important to identify the possible attack propagation within a software architecture from a given starting point.

The last aspect is the identification of attack paths to selected architectural elements. For instance, in our running example, the `ProductStorage` component stores highly confidential data. Losing the data managed by this component is the worst-case scenario in our example because it means that others have access to the blueprints of P's products. Therefore, software architects want to identify whether there exist attack paths leading to the `ProductStorage`. In addition, they are particularly interested in attack paths leading from the externally accessible `Terminal` to the `ProductStorage` because the `Terminal` is accessible by an external person. This external access makes it easier to compromise a component. Identifying these potential attack paths is important to mitigate them effectively.

We will now summarise the aspects of our running example and how they relate to our contributions. Firstly, we analyse usage and malicious usage with respect to access control policies, which is our contribution C3. Secondly, we address the propagation of attacks through the contribution C4.1. Lastly, the identification of attack paths is addressed by our contribution C4.2. In

the following sections, we will use these aspects to describe and present our contributions in detail.

Architectural Element	**Attributes**	**Vulnerabilities**
Access Terminal Services	Technician, Failure, S	—
Storage Server	Admin	—
Terminal Server	Admin	CVE-2021-28374
Machine Controller	Admin	—
Access ProductionDataStorage Services	Machine	—
Access ProductStorage Services	ProductDeveloper	—

Table 3.1.: Access control policies and vulnerabilities for the running example

Part II.

Contributions

4. Modelling Influencing Factors for Context-Based Security

During our work, we investigated various factors that influence context-based security on the software architecture. Our investigation focused on confidentiality and its relationship to software architecture. The factors may not be complete, as we did not perform a systematic analysis, but they are essential to our analyses and influence our results. We derived the influencing factors based on our involvement in different research projects. These projects covered the Industry 4.0 domain with the projects *Trust 4.0*[1] and *FluidTrust*[2], the smart grid domain with the *Smart Grid Resilience Frameworks*[3], and the mobility domain with the KASTEL Mobility Lab. During these projects, we identified different scenarios and use cases [12, 11, 169] and identified related work. Based on these cases and related work, we identified the influencing factors. The concrete justification for the relevance of each influence factor can be found in the respective modelling sections. Our influencing factors are the following:

- Access control for protecting architectural elements (c.f. Section 4.2)
- Vulnerabilities affecting architectural elements (c.f. Section 4.3)
- Attacks exploiting vulnerabilities (c.f. Section 4.4)
- Attackers with the capability to use attacks and access control properties (c.f. Section 4.5)

[1] *Trust 4.0 Dataflow-based privacy for industry 4.0.* Jan. 19, 2019. URL: https://web.archive.org/web/20210422205559/http://trust40.ipd.kit.edu/home/ (visited on 06/06/2023).

[2] *FluidTrust - Enabling trust by fluid access control to data and physical resources in Industry 4.0 systems.* Aug. 8, 2020. URL: https://web.archive.org/web/20220525001447/https://fluidtrust.ipd.kit.edu/home/ (visited on 06/06/2023).

[3] *Smart-Grid-ICT-Resilience-Framework.* URL: https://github.com/kit-sdq/Smart-Grid-ICT-Resilience-Framework (visited on 06/06/2023).

Architects should be able to express and consider these factors during the architectural design. However, it is challenging to track these factors and assess their impact in larger systems. Automatic architecture-based security analyses considering these factors could help architects to determine the impact of these factors on the architecture. For archiving these analyses, a well-defined model is necessary. Architects can already use ADLs such as the PCM to define an architecture with a well-defined syntax and semantics. For our influencing factors, we also wanted to provide this possibility and, therefore, extended PCM to support our influencing factors. As discussed in Section 2.1, PCM provides the possibility to model software architectures with components and interfaces as well as the hardware environment with processing resources and network resources. Therefore, it is a suitable candidate. Additionally, PCM already supports other architecture analyses, and we have more experience using it compared to other ADLs. The integration in PCM enables architects to annotate existing software architecture elements with our factors. Nevertheless, insights gained from our modelling and modelling principles can be transferred to other ADLs like UML. In addition, this annotation will provide some documentation for security properties. This chapter builds up on the metamodels initially proposed in our publications [211, 214, 94, 209, 212].

The remainder of the chapter is structured as follows. We explain in Section 4.1 an extension to identify service and component instances within PCM. This is necessary because otherwise, we cannot later annotate these elements with access control properties and vulnerabilities. We describe the access control properties in Section 4.2. This includes the metamodel for access control policies and its integration into PCM. The vulnerability metamodel is described in Section 4.3. The attacks exploiting these vulnerabilities are described in Section 4.4. The attacks an attacker can perform represent the capabilities of attackers. We describe the attackers in Section 4.5.

4.1. Identification of Services and Components

This section is a prerequisite for our contributions C1 and C2. It introduces new model elements for PCM to identify services and components for annotating access control policies and vulnerabilities. Access control policies can regulate the access to services, such as in our running example, the service

of the `ProductionDataStorage` component to store the log of the `Machine`. Another example is the SSH service of a server. Usually, each service instantiation has a custom access control policy specific to its system. For instance, the `ProductionDataStorage` component could be used in another system that stores log data, but it would be a different machine. The access control policy should not be the same in both systems. It needs to differ from the specified policy in our running example. Moreover, using the same access control policy is even a security risk, similar to CWE-798 [116], where hard-coded credentials are used. Therefore, it should be possible to specify access control policies system-specific. This leads to the fact that we need architectural elements for identifying access control policies for system-specific services.

This need is further increased with the consideration of vulnerabilities. Services can contain vulnerabilities. However, as is the case with access control policies, not every instantiated service type might be vulnerable to the same vulnerabilities, nor does every vulnerability have the same impact on the service type. For instance, considering the vulnerability of the running example, the NVD's description states that it "may include a cleartext password in some configurations" [129]. Therefore, there is a need to identify system-specific the vulnerability since the impact may vary depending on the specific configuration for a service. Moreover, in certain cases, a vulnerability may not even be relevant to the service type at all. Therefore, it is important to identify the specific vulnerable service in order to determine the impact of a vulnerability.

Both parts, the access control policies and the vulnerabilities, are essential to our approach and both require the service identification to be system-specific. This need for system-specificity can also be applied to the specification of components. In our modelling approach, we use components to abstract from certain details. For instance, we might not model maintenance services but only assign an access control policy to a component and use the policy to represent a maintenance service. Therefore, we also need to identify components in a system-specific way.

The PCM supports modelling of services and components in the system independent repository [154, p. 44f]. Software architects can instantiate the components as an `AssemblyContext` in the system-specific assembly [154, p. 44f]. By instantiating an `AssemblyContext`, the services associated with the component are also implicitly instantiated. While we can annotate the `AssemblyContext` with our access control and vulnerability extension, the PCM

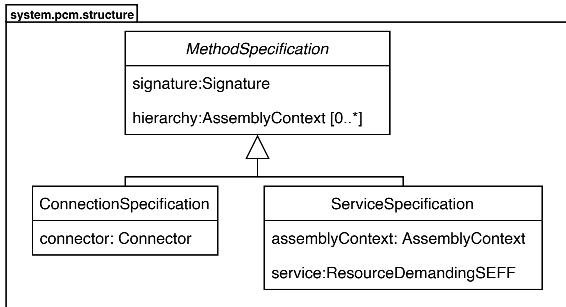


Figure 4.1.: Metamodel elements to identify services for annotation

lacks a metamodel element to explicitly identify the instantiated services. As a result, our modelling extensions are unable to annotate services with access control policies or vulnerabilities leading to potentially incorrect results.

To enable the annotation of instantiated services, we extended PCM with new elements specifically designed to identify the instantiated services. Based on the PCM definition from Reussner et al. [154, p. 44f], we consider the new elements to be structural and system-specific. Figure 4.1 illustrates our identification elements in the package `system.pcm.structure`. We distinguish between two main elements, the `ConnectionSpecification` and the `ServiceSpecification`, which both share the same common abstract parent class `MethodSpecification`. The `MethodSpecification` encapsulates the `Signature` of the service and its hierarchy is decoded as an `AssemblyContext` list. The hierarchy serves as a special workaround for `CompositeComponents` in PCM, which are components that consist of multiple subcomponents. These `CompositeComponents` and the assembly (system model) are modelled similarly in PCM. This simplifies the modelling and development effort in the repository. However, it has the disadvantage that subcomponents of components can only be identified through their hierarchy. For a subcomponent, the hierarchy would contain the parent instantiated component. For multiple nested subcomponents, this would result in a list of parent components. This modelling technique is common in PCM analyses and similar metamodel extension with hierarchies of `AssemblyContexts` lists can be found in other PCM projects, such as the data flow analysis [165].

With the currently described attributes, we cannot yet fully identify an instantiated service. The signature only shows the name of the service, but not the actual instantiation and the hierarchy only identifies a potential parent component. The actual identification is done by the two subclasses. The `ServiceSpecifiation` adds the concrete instantiated component with the `AssemblyContext` and adds the references to the `ResourceDemandingSEFF`. This enables us to identify the concrete service as we now have the name, the potential parent component, the actual instantiated component and the implementation. This is sufficient information to identify a service in our scenarios. However, in some cases, we are interested not only in which service is called but from which component this happens. For instance, if the component which calls a service is trustworthy, there might be a less restrictive policy in place. This feature is realised by the `ConnectionSpecification`. It contains only a `Connector`. In PCM, this connector contains enough information to infer the other necessary properties to identify the concrete instantiated service.

In summary, we can use these new model elements to identify instantiated services in order to annotate them with access control policies or vulnerabilities.

4.2. Considering Access Control Properties in Software Architecture Models

This section answers our research question **RQ1.1** by providing an access control metamodel. It is our contribution C1 and enables security experts to model access control policies within a software architecture. Besides enabling the modelling of access control policies, the metamodel provides support to model further access control properties, such as the used attributes. In addition, it provides the foundation to answer **RQ1.2** and **RQ2**.

Access control policies are important for ensuring confidentiality because they provide the means to regulate the access to architectural elements. Therefore, they provide protection against unauthorised or malicious users. While access control is not the only mechanism for protecting information, it is a widely used and effective approach. Other mechanisms, such as usage control [138] or encryption [146], can also preserve the confidentiality of

data and can be used in combination with access control policies. Many systems today use some form of access control, such as Windows with their user management or large cloud systems, such as Google login services for accessing Gmail. Therefore, considering access control in the architecture is beneficial. However, access control policies and their associated properties need to be expressible so that architects can define policies and analyses can evaluate them. Therefore, we developed a metamodel for expressing access control policies and their properties.

In this section, we first describe our requirements in Section 4.2.1, which guided the development of the metamodel and then discuss our attribute metamodel in Section 4.2.2. Afterwards, the attributes are used in the access control policies which are described in Section 4.2.3. In Section 4.2.4, we describe the integration of attributes to describe the context in PCM. This integration is necessary to describe the system environment and enable context switches within a software architecture. After describing the access control policy and the context description, we explain in Section 4.2.5 the transformation from our access control model to a valid XACML model and how access requests are handled. The transformation and request generation are the foundation for the later analyses.

4.2.1. Requirements for Modelling Access Control Properties

During our work, we identified several requirements for an access control metamodel. These requirements guided the metamodel development. We derived them iteratively based on our experience in different research projects and case studies. Our initial starting point was the use cases and scenarios that we gathered in collaboration with our industrial partners in [11, 169, 12]. Over time, we continuously refined and extended the initial requirements. We will first list the requirements and afterwards explain our rationale for each one. We derived the following requirements for our access control metamodel.

AR1 Security experts should be able to specify access control policies for the logical and infrastructure layer.

AR2 Security experts should be able to define access control policies which can consider the context during policy evaluation.

AR3 Security experts should be able to specify arbitrary attributes for access control policies and not restricted to predefined classes.

AR4 Security experts should specify access control policies based on familiar concepts.

AR5 Security experts should be able to reuse access control policies during runtime.

AR1 describes the differentiation between the logical and infrastructure layers. While existing approaches, such as Seifermann et al. [168], already consider access control at the logical layer, they do not consider the infrastructure layer, such as the hardware resources. We define the logical layer in PCM as the invoked services of an instantiated component or the used data. For instance, in the running example, this is the technician's access to the log data by accessing the services provided by the `Terminal`. However, the software architecture consists of more than just service calls and data. In PCM, there are also the `AssemblyContexts`, `ResourceContainers` and `LinkingResources`. Each of these elements can have separate access control policies for maintenance. For instance, the `LocalNetwork` in our running example represents a network component like a router or switch. These elements often have a separate access control policy for maintenance. An example where both layers are necessary is our running example. The running example is based on a described scenario in Al-Ali et al. [12]. It contains access control policies for services, such as the `Access` service at the `Terminal`. In addition, it contains maintenance access to the components themselves, such as for the `MachineController`. Therefore, it is necessary to consider both layers. Another reason to consider the infrastructure layer in the access control model is the attack analyses. Attackers use the infrastructure to propagate within a system like in Advanced Persistent Threats (APT) [40]. There, attackers often use other infrastructure services like phones or printers to perform lateral movements within an attacked system.

In Al-Ali et al. [11], we listed, together with our partners, different use cases and requirements for Industry 4.0 systems. One key aspect that emerged from this work is that the access is not granted based solely on statically assigned roles but depends on the dynamic attributes, such as the time or the state of a machine. Hence, we derived the requirement **AR2**. It states that security experts can formulate access control policies which consider the system context during the policy evaluation. The context of a system can be described as attributes, such as the location of workers. These attributes can

dynamically change, such as in the case of the location. Hence, the context consists of dynamically changing attributes. Similar attributes are found in our running example. For instance, the access to the log data depends on the machine's state. The machine's state can be seen as an attribute of the machine and can vary during the runtime. Also, Bertino et al. [29] describe the need for "richer access control models" [29]. These additional attributes can be described as context. Also, other approaches, such as OrBac [43] or Zhang et al. [225], consider the context for the access control policy. In our case, we restrict the model and analysis to policies with dynamically changing attributes. Like in the mentioned approaches, the consideration of the context does not require the policy itself to be dynamic. The policy itself can be static. This is similar to our definition in [215], where we defined dynamic changes for confidentiality. This definition can be transferred to access control policies, which results in access control policies specified during the design time considering dynamic attributes and, thereby, the context of the system.

In Boltz et al. [33], we developed a metamodel which fulfils our previous requirements, but not **AR3**. In the publication, we extended an existing confidential analysis [167] with a context-based model. We had predefined context classes, such as for locations or roles. These context classes described the context and were considered in our initial analysis. However, the predefined classes also limit the approach to the initial context attributes. While modelling different use cases, we wanted to extend the approach to other context attributes. The consideration of additional context classes, such as the time required a metamodel change. Security experts would need to change for every new context type the metamodel. However, this is very inflexible, especially in a dynamic environment like in Industry 4.0. Other related models, such as ABAC [74], have a more flexible model to define different context types. Here, the context attributes can be directly described within the model, and no metamodel changes are required. Therefore, our final model should provide similar behaviour.

AR4 requires that the developed metamodel uses familiar concepts for security experts. We consider a concept familiar if the abstract syntax and semantics are similar to other metamodels. Specifying access control policies is complex [204] since they consider multiple different dynamic attributes and values. Therefore, it is beneficial if our new metamodel follows familiar concepts, so security experts can more easily use them because they are already familiar with the abstract syntax and semantics of elements. For

instance, if security experts already know the meaning of an element, they do not need to look up the meaning.

The last requirement **AR5** describes that the modelled policies should be reusable during runtime. With the help of our analysis, architects can analyse the impact of access control policies and find satisfying policies for their use cases. After finding satisfying policies, the policies need to be integrated into the actual runtime system. However, many unintentional errors can happen in a manual transformation from the design time access control model to the runtime access control model. This can lead to policy misspecifications. For instance, a possible scenario for a misspecification is simple typos, such as *admn* instead of *admin* or the usage of synonyms, such as *root* instead of *admin*. In addition, the manual transformation is very cumbersome. Hence, it is beneficial to avoid mistakes and reduce effort, to reuse the analysed policies during runtime without manually transforming the access control policies to the runtime policies.

4.2.2. Modelling Attributes

The access control decision is usually based on some kind of attributes a requestor has. In classical approaches like in RBAC, the types are limited to a restricted set, such as the role in RBAC. Also, our initial prototype [33] used something similar. However, as in **AR2** described, our access control model should provide a more flexible definition of attributes. This allows us to better adapt the access control policies to the actual context required for accessing an element without changing the metamodel itself.

The idea of using attributes to describe the context is not new. Other approaches, such as Zhang et al. [225] or especially ABAC [74], use similar concepts. In our case, we choose to transfer and extend the attribute modelling concept of the ABAC implementation XACML to the software architecture. XACML uses a generic way to model attributes and supports a wide variety of datatypes. In addition, it is well documented and a widespread standard. Therefore, it also fulfils our **AR4** since architects might already be familiar with the concept.

Figure 4.2 illustrates our attribute model. The grey elements are the original elements from XACML. The attributes in our metamodel are mainly encapsulated in the package `systemcontext`. Software architects can specify

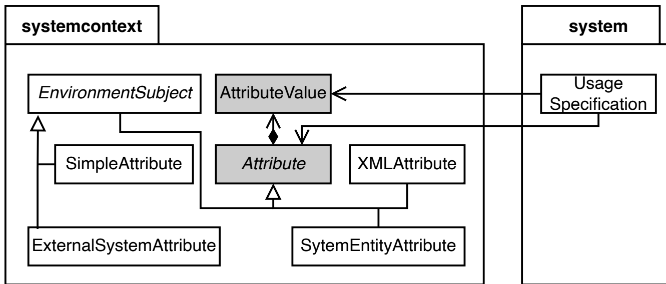


Figure 4.2.: Attribute metamodel with gray elements based on XACML

similar to the type in programming language an `Attribute`. For instance, in our running example, the role is a type. The concrete instantiated attribute is then the `AttributeValue`. Besides the value, it also contains some additional metadata, such as the actual datatype used to represent the values, such as String or Integer. An example value from our running example can be the value `Maintenance` for the role of an employee. The `UsageSpecification` can act as a trace element between the modelled attribute type and the attribute value. These elements are used later in the other metamodel parts to reference to attributes with concrete values. A `UsageSpecification` always has one `Attribute` and an optional `AttributeValue`. However, usually, both exist for a `UsageSpecification`. The optionality is considered as a special case for the `XMLAttribute`. The `Attribute` is an abstract element for better integration into the software architecture. We differ between `XMLAttribute`, `SystemEntityAttribute`, `SimpleAttribute`, `ExternalSystemAttribute`.

The `XMLAttribute` enables architects to write XACML-based statements in XML. These statements are then parsed during the transformation (c.f. Section 4.2.5) and integrated into the generated XACML policy file. This enables architects to use custom XACML attribute definitions. One benefit is that architects can use custom-defined datatypes and are not restricted to our implemented datatypes. This also helps to solve **AR3**. In addition, if architects are already familiar with XACML, they can directly insert XACML statements. However, the drawback is that they have to consider all the aspects necessary for the PCM integration themselves. For the other `Attribute`'s children, the transformation and analyses automatically consider these.

In certain cases, knowing which system entity issues an attribute is important. For instance, the error state of the machine in the running example can be provided by an entity. However, a malicious user can exploit this behaviour by providing its own state and thereby avoiding the protection mechanism. For avoiding this security issue, XACML defines the issuer attribute for attributes. The issuer defines from which element the attribute must be issued.

We modelled this behaviour with the `SystemEntityAttribute`. It links an attribute to an `Entity`. `Entity` is in PCM the superclass for various architecture elements. This enables us to link attributes, for instance, to `AssemblyContexts`, `ResourceContainers` or `LinkingResources`. Therefore, we are not restricted in the assignment to one layer, such as the logical and can assign attributes also to others as we described in **AR1**. In our running example, we model the machine state as a `SystemEntityAttribute` and select as `modelEntity` the `Machine` representing the issuer.

Attributes which do not have an issuer represented in the architecture are modelled by the abstract `EnvironmentSubject`. Of course, in a fully modelled software architecture, each attribute needs to be created by some element in the software architecture. However, having a fully modelled software architecture can have drawbacks, such as a high modelling effort and not a good overview. In some cases, this additional insight from high details is not useful. Therefore, abstraction is used to only model the important aspects. For instance, in our running example, we have no dedicated components for providing access control mechanisms. Yet, we can still analyse the system regarding certain access violations. In these cases, where software architects just want to represent such attributes, the `EnvironmentSubject` can be used. An `EnvironmentSubject` also has a boolean flag to indicate whether the attribute is assigned to the subject or environment (c.f. Section 2.2.1). The `SimpleAttribute` models attributes with no specific further relationships. For instance, in our running example, we model the Role with a `SimpleAttribute` since it has no dedicated issuer from the system or external relations. In case an attribute should have a relationship to a dedicated issuer, but this issuer is not modelled within the architecture as a dedicated element, the `ExternalSystemAttribute` can be used. The issuer is then set by the `externalName`.

Each `AttributeValue` is assigned a datatype for the stored `values`. Regardless of the datatype, each value is serialized as a String. The transformation and editor for the metamodel must handle the deserialization and serialization.

Our metamodel only supports a subset of the datatypes from XACML. We currently support String, Boolean, Integer, Double and Date. However, the other types can be used in the `XMLAttribute`. Therefore, we still have no limitation to predefined classes. In addition, these additional datatypes can be added to the metamodel. In this regard, a developer, extending the metamodel, would need to adjust the datatype enum by adding the new datatype and then add custom handlers for the editors and the XACML transformation.

In our thesis, we also reuse the attributes as representations of credentials. For instance, we assign the roles of a user also as an attribute. Usually, in a system, the user would enter their credentials, and then the system would assign the role. The entering of the credentials is the authorisation of the user against the system. After the authorisation, the system knows the user and can assign them attributes. Therefore, there is a connection between the authorisation process and the assignment of attributes. We abstract from the authorisation process during our thesis and assume that the authorisation process is given by the role assignment. Hence, we use attributes as a synonym for credentials in the case of user roles. The meaning is that a user with an assigned attribute, like the role, has the ability or knowledge to get the authorisation for the role. For instance, based on our running example, we assign the role technician to a user. This would also mean that the user has the ability to get this role. The concrete authorisation mechanisms, such as credentials with username and password, are not modelled.

4.2.3. Modelling Access Control Policies

Access control policies are essential for managing access to assets that require protection. They regulate in which cases entities are granted access, or access should be denied. In approaches, such as ABAC, access is granted based on the current attributes that a requestor possesses. These attributes can vary depending on the circumstances, such as with the machine state in our running example, and describe the context. Therefore, using an approach like ABAC would fulfil our **AR2**. In addition, ABAC does not limit the attributes to certain predefined classes. Therefore, arbitrary types of attributes can be used. This would also match our **AR3**. Also, ABAC is not limited to a particular layer and can be used for infrastructure or logical aspects. So **AR1** is also fulfilled. However, the ABAC concept in itself is only theoretical. Therefore, we would need to develop an entirely new metamodel, which

would not be very familiar to experts. This would violate **AR4**. However, there are industrial implementations, such as XACML. In our case, we base our access control model on XACML.

Figure 4.3 illustrates the policy part of our access control metamodel. The grey model elements are semantically similar to their counterparts in XACML. The white elements are new elements that are relevant for the integration into PCM. For the white and grey element (`AllOf`), we slightly changed the syntax and semantics compared to XACML. The syntax of all figures is based on UML class diagrams. The aggregate symbols indicate a containment relationship in the metamodel.

The metamodel for the policy specification is mainly contained in the `policy` package. The package itself is decoupled from the PCM and could therefore be reused for other ADLs. All access control policies are contained within a `PolicySet`, as in XACML. It helps to group different policies together. For example, it allows all policies related to a component or use case to be modelled within one set. This can improve readability by allowing related policies to be grouped together. A `Policy` is contained by a `PolicySet` and contains different `Rules`. The `Rule` encapsulates the actual access control rule with its `Expression`. The target to which a `PolicySet`, `Policy` or `Rule` applies is selected by the `AllOff`. Unlike XACML, we have omitted the actual target element in our metamodel, as it is not necessary in our cases. An element may have multiple `AllOf` elements if they are applicable to multiple targets. The `AllOf` acts as a logical disjunction in this case. For example, in our running example, the access control policy for regulating access to the `StorageServer` and `TerminalServer` is realised with a `Rule` containing two `AllOf` elements.

The actual target is modelled by a list of `Match` elements, which form a logical conjunction. This means that for an `AllOf`, all matches must be fulfilled. Otherwise, it is not applicable. `Match` is abstract to decouple the policy metamodel from PCM. Therefore, the four concrete matches are logically separated in the `structure` package, which encapsulates the PCM integration. The concrete matches provide a better integration for the PCM elements in our access control metamodel. The first element is the `EntityMatch`. It is used to select an entity. For instance, in our running example, this could be the `StorageServer`. However, it can also be used for components, enabling us to select targets for different layers as required in **AR1**. It is also open for extension to new architectural elements as long as they also use the common superclass. The

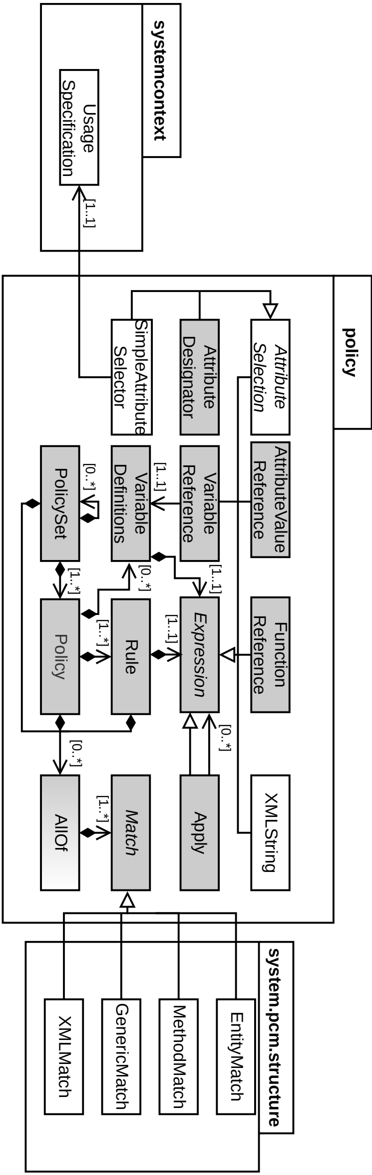


Figure 4.3.: Simplified access control policy metamodel with grey elements for elements based on XACML and white elements as new elements[211]

second match type is the `MethodMatch`. It models the selection of a service by containing a `Methodspecification` (c.f. Section 4.1). The access service provided by the `Terminal` in the running example can be, for instance, selected by using a `MethodMatch` and adding a `ServiceSpecification`. The `GenericMatch` can be used for selections based on attributes. It is useful when the previous two matches are too restrictive, but not the complete flexibility of a custom XACML match is necessary. It resembles the XACML's matching definition but uses our metamodel elements. Software architects need to select a matching operation from the boolean operations from XACML and in which category the attribute is found. In addition, they must select an attribute value. In case a software architect needs the full flexibility of XACML with custom-defined operations or attributes, the `XMLMatch` can be used. Similar to the `XMLAttribute`, the `XMLMatch` is used to specify directly XACML statements. These are saved as a String formatted as XACML.

Depending on the target selection, the result may not be a single element but multiple elements. Therefore, a policy can have multiple `Results` and a `PolicySet` can have multiple `Policy` elements. However, each can have a different decision regarding the access control. Therefore, combining algorithms are used as in XACML. These can be assigned to the `PolicySet` and `Policy`. They reduce multiple access decisions to one decision. We support in our metamodel all of the combining algorithms from XACML. For example, one combining algorithm is *DENY_UNLESS_PERMIT*. It states that the request will always be denied unless an applicable policy explicitly returns permit. The opposite combining algorithm would be *PERMIT_UNLESS_DENY*, which always returns permit unless a policy denies the access.

The actual access control decision is encoded in the `Rule` by the attribute `decision`. In our metamodel, we simplified it to `Permit` and `Deny`. However, in XACML, more values like `not applicable` exist. The other types are mapped to `Deny` during the analyses. These additional elements were unnecessary for our cases, but the metamodel could be easily extended by adding them to the `PermitType` enumeration and considering them during the transformation. The access condition is encoded in the `Expression`. XACML defines here multiple different subtypes. We support eight different elements, which are mostly based on their XACML counterparts. Like for the attributes and the match elements, the `XMLString` models a custom XACML element, expressed as a String. It is used for custom extensions. The `Apply` performs an operation. The parameters of the operation are again `Expressions`. For instance, in our running example, the check for the two attributes of the

service technician can be modelled by an `and` operation and, as parameters, two selection operations for the expected attributes. The `FunctionReference` contains a reference to a function, and similarly, the `VariableReference` contains a reference to a `VariableDefinitions`. The `VariableDefinitions` contain an `Expression` and are contained by a `Policy`. They can be used to define reusable `Expression` parts. The `AttributeValueReference` is used to reference to an `AttributeValue`. The abstract element `AttributeSelection` is used as a parent class for all elements, which select attributes from the request. There, the software architect first defines the category of the attributes in the request. The `AttributeDesignator` is then used to select the concrete attribute. In XACML, then a bag with the attribute values is returned. However, for selecting then a concrete attribute value, like in our running example, the `Admin`, software architects would need to model functions accessing the bag and then comparing the value of it to the desired attribute value (`Admin`). Since the use case is in our cases very common and the modelling effort is very high for a simple attribute comparison, we added the `SimpleAttributeSelector`. This element simply compares a request attribute against a predefined `Usagespecifications`. This simplifies the use case for an attribute comparison since a software architect only needs to use this element. However, if software architects do not want to use the `SimpleAttributeSelector`, they can also use the way with the `AttributeDesignator`. During the analyses, we transform the `SimpleAttributeSelector` back to XACML statements.

Listing 4.1 illustrates an access control policy for our running example as a textual representation. The textual representation is a simplified presentation similar to JSON [78] for the actual model. For instance, it does not contain the unique IDs for elements like the `PolicySet`. For simplicity reasons, it contains only the policy where the service technician tries to access the log-data of the machine.

The policy first contains the definition of the `PolicySet` with the selection of combining algorithm. As the combining algorithm `DENY_UNLESS_PERMIT` is used. As previously explained, this states that access is always denied unless it contains at least one `Decision` with `Permit`. Then a `Policy` is also defined with a similar combining algorithm. Starting from line 5, the `Rule` is defined. It contains a name for the `Rule` and the `decision`. Here, it is `permit`. This selects that if the condition is true, the `Rule` returns `permit`. If we change it to `deny`, the `Rule` will return `deny`. The access condition is encoded starting from line 8 with the `Apply`. In our example, the condition should check for two

Listing 4.1: Simplified textual policy representation of the access control policy for the technician in the running example based on Walter et al. [211]

```
PolicySet {
    combining : DENY_UNLESS_PERMIT,
    Policy {
      combining : DENY_UNLESS_PERMIT,
      Rule {
        name : "Technician with Machine failure",
        decision : permit,
        Apply {
         function : and,
          SimpleAttributeSelector {
           UsageSpecification {
            attribute : role,
            value : technician,
            }
          },
          SimpleAttributeSelector {
           UsageSpecification {
            attribute : machineState,
            value : failure,
            }
          }
         }
        }
        AllOf{
          MethodMatch{
            ServiceSpecification{
                assemblyContext : Machine,
                Signature : Access,
                SEFF : RDSEFF_Access,
            }
          }
        }
      }
    }
}
```

attributes, that the accessor is the technician and that the machine is in the error state. We realise it by selecting `and` as the `Apply` function. It represents that the values of the parameters form a logical conjunction, meaning that both parameters must return true for the condition to be true. The parameters are, in our case, functions to check for the two attributes. Since, in our case, the attributes are quite simple, we do not have to use complex conditions.

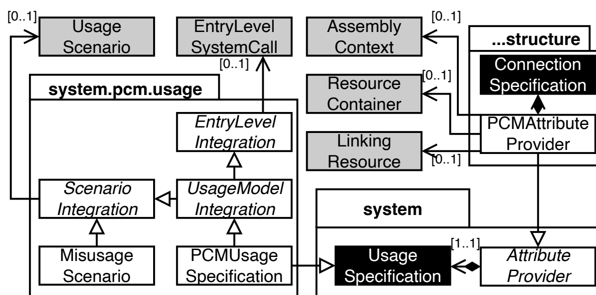


Figure 4.4.: Attribute provider and scenario metamodel for context-based policies

We can use the `SimpleAttributeSelector` as parameters (l. 10 and 16). In each `SimpleAttributeSelector` we define a `UsageSpecification` with an `attribute` and a `value`. For instance, for the first parameter (l. 10ff), it is the `role` attribute and the `technician` as value. The last part of the example policies contains the target selection with the `AllOf` starting from line 24. In the example, the policy should address the service call on the machine from the technician. Therefore, we use a `MethodMatch` for selecting a service. Our example uses the `ServiceSpecification` for selecting the concrete service. The service is identified by the signature (`Access`) representing the name, the `AssemblyContext` (`Machine`) representing the instantiated component and then the `SEFF` representing the concrete implementation.

4.2.4. Modelling Attribute Providers and Scenarios

So far, the metamodel covers the modelling of access control policies and the modelling of attributes. We also need to describe the requestors and their attributes for analysing the specified policies. The requestor can be a user or other automatic system calls, such as a cron job [141] or delegated system calls. Sometimes the attributes can also be extracted from the modelled system entities, such as in our running example, the machine provides its state as an attribute or the cron jobs run with a certain role. We will first describe the assignment of attributes to users and then the concept of system elements providing attributes.

Figure 4.4 illustrates our integration of attribute modelling in scenarios and for architectural elements providing these. The grey elements are the existing PCM elements, the black elements are concepts we described previously and the white elements are new concepts. The metamodel is split over three packages (`system.pcm.usage`, `system`, `system.pcm.structure`) to differentiate between the general concepts (`system`) and their integration in `PCM`. For the assignment of attributes to users, we decided to reuse the concepts in PCM for usage modelling. PCM has the `UsageScenario` to model the intended usage of a system as a scenario. It consists, among other elements, of `EntryLevelSystemCalls` representing the system call of a user. This assignment concept is initially published in Walter et al. [214].

The idea is that we add the attributes to a scenario and a system call. The attributes are describing then the context of the system and the user. The system context can be, for instance, the state of the machine in our running example and the user context can be, for instance, the role. We provide an abstract integration element for each element: the `EntrylevelIntegration` and `ScenarioIntegration`. They each reference the corresponding PCM element. The abstract `UsageModelIntegration` inherits from both (`EntrylevelIntegration`, `ScenarioIntegration`). The actual model which sets the attributes is the `PCMUsageSpecification` it inherits from the `UsageModelIntegration` and the previously described `UsageSpecification` (c.f. Section 4.2.2). From the `UsageModelIntegration`, the integration into `EntrylevelSystemCalls` or `UsageScenarios` are inherited. The attributes are inherited by the `UsageSpecification`. The modelling approach allows specifying the context for a scenario and for the system call, which can result in ambiguous context specification since a `UsageScenario` can have different attributes than its containing `EntryLevelSystemCall`. In our case, we decided that the attributes of a `EntryLevelSystemCall` completely remove the context in which the `UsageScenario` is contained. This enables an architect to model context changes within one `UsageScenario`. This is useful, if two users work together during one usage scenario.

Besides analysing the intended usage, the consideration of the usage from malicious users is important. Similar to the concepts of *misuse cases* [177] and *mal-activity diagrams* [176], we define a misusage scenario. These are scenarios which should not be possible. Therefore, access control policies should somehow prohibit them. We model these scenarios with the `MisusageScenario` element. This enables an architect to just select an `UsageScenario` and thereby declare it as a misusage scenario.

So far, the metamodel allows the assignment of attributes to a user or the overall system context during a usage scenario. In some cases, it is beneficial to have the capability to assign them to other architectural elements. For instance, in our running example, the machine provides its state. Representing this aspect also in our architectural model enables our analyses to consider this aspect, for instance, if a component is compromised. Another aspect is that during a service call, the context switches. For instance, in our running example, the machine uses its role to store the log data on the storage server and does not delegate the role of the user. In our modelling, we call this concept attribute provider since the annotation of architectural elements indicates that the annotated elements provide certain attributes. This concept was initially published in Walter et al. [211]. In our metamodel, this is represented by the abstract `AttributeProvider`. It contains an `UsageSpecification` representing the provided concrete attribute value. The `PCMAttributeProvider` is the child of the `AttributeProvider` and provides the concrete PCM integration. This inheritance between the concrete PCM-specific element and the abstract elements should help for a better extendability for other ADLs. Currently, we can assign attributes to four PCM elements. The first three elements are PCM elements. These are `AssemblyContext`, `ResourceContainer`, and `LinkingResource`. Each element is referenced. In addition, the `PCMAttributeProvider` can contain a `ConnectionSpecification`. Other examples for the usage of attribute providers besides the state in the machine could be saved credentials in a device or the domain controller in a network providing user roles. These cases are particularly useful for structural, architectural elements, such as the `AssemblyContext` or `ResourceContainer`. The combination of a `PCMAttributeProvider` with a `ConnectionSpecificiation` is especially useful to model context changes within a system call. For instance, a system call connects to a database, and the database has different credentials. Then a software architect can assign the connection between these two architectural models new attributes. In our running example, we use this to assign the role `machine` to the connection between the `Machine` and the `ProductionDataStorage`

4.2.5. Transformation to XACML & Access Requests

Besides the specification of access control policies, we must also evaluate the policies regarding an access request. However, there are two kinds of policy evaluation necessary. The first is during our analyses and the second

is during the runtime as described in **AR5**. We could implement our own PDP interpreting our access policy model for the first part. The same could be done for the second part and provided as a library for runtime systems. However, especially for the second part, it would require projects to use our PDP and therefore limit the practicability since many existing projects already have implemented PDPs. In addition, the implementation of a custom PDP is complicated and could be error-prone since it requires excessive testing and the consideration of many edge cases. Therefore, we choose to reuse an existing XACML-based PDP. In our case, we use the XACML reference implementation from AT&T[4] since it is open source and, therefore, easily integrable in our analyses. Despite the fact that our access control policy metamodel is similar to XACML, we cannot use the PDP directly because of our custom metamodel elements, such as the `SimpleAttributeSelector` or the `EntityMatch`. Therefore, we need to transform our access control policy model to a valid XACML model. This transformation is also beneficial since it allows us to reuse the policies during runtime (see **AR5**) and to replace the used PDP with any other XACML-based PDP. We checked the validity of the transformed files by comparing the generated XACML policies against the XACML schema file [134] provided by OASIS. We use the XACML 3.0 standard [132]. We compared the use cases from our usage analysis evaluation (c.f. Section 7.1) using the schema file with xmllint[5].

Figure 4.5 illustrates the transformations into XACML. We differentiate between the *policy transformation* and the *access request transformation*. The first one transforms the access control policy model from Section 4.2.3 into a valid XACML file. This file can then be loaded by a PDP during our analyses or used in a runtime system. The second part transforms requests from our analyses to a valid XACML request. This request is then sent to a PDP and evaluated. The result is then transformed back into a model for our analyses. We first explain the *policy transformation* and later explain the *access request transformation*.

The policy transformation works similarly to *Top-Down Parsing* [6]. It starts from the most outer `PolicySet` and then transforms each child element in the abstract syntax tree. After the transformation to XACML, we serialize the model as an Extensible Markup Language (XML) file by using the Java-built

[4] AT&T Open Source. URL: `https://github.com/att/xacml-3.0` (visited on 10/25/2021).
[5] *libxml2*. URL: `https://gitlab.gnome.org/GNOME/libxml2` (visited on 01/09/2023).

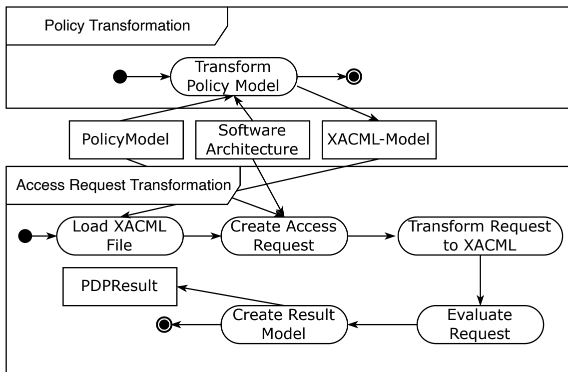


Figure 4.5.: Process for creating a XACML model and creating an access request

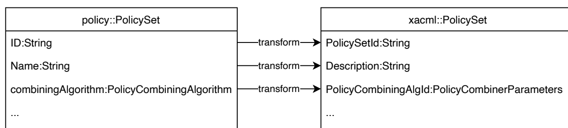


Figure 4.6.: Excerpt PolicySet Transformation

in XML mapping with the *Marshaller*[6]. Listing 4.2 illustrates the serialized XACML model for the Listing 4.1 based on our running example. The transformation is mostly straightforward for all the XACML-based metamodel elements. We take the policy model element and create the corresponding XACML element. Figure 4.6 shows an excerpt of the `PolicySet` transformation. On the left side, the `PolicySet` from our approach is listed, and on the right side, an excerpt of the `PolicySet` from XACML is listed. This is also indicated by the namespace prefix with `policy` for our element and `xacml` for the XACML element. The `ID` is transformed to the`PolicySetId`. The `Name` is the `Description` and the `combiningAlgorithm` to the `PolicyCombiningAlgId`. The three dots indicate the other attributes from the `PolicySet`. They can be transformed accordingly. In Listing 4.2, the serialized output for the policy set is illustrated. In the following, we will explain the transformation in more detail.

6 *Interface Marshaller*. URL: `https://jakarta.ee/specifications/platform/9/apidocs/jakarta/xml/bind/marshaller` (visited on 01/09/2023).

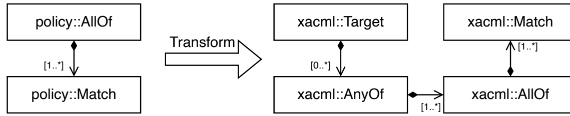


Figure 4.7.: AllOf transformation

$$
\begin{aligned}
xacml::Target &\rightarrow \epsilon \mid xacml::AnyOf^{+} \\
xacml::AnyOf &\rightarrow policy::AllOf \\
policy::AllOf &\rightarrow xacml::AllOf\ policy::Match^{+} \\
policy::Match &\rightarrow xacml::Match^{+}
\end{aligned}
$$

Equation 4.1.: AllOf transformation rules

The first element with a slightly changed syntax and semantics is the `AllOf`. Figure 4.7 illustrates the difference structures. On the left side is an excerpt of our policy metamodel, and on the right side, the XACML metamodel. However, since the overall semantics for the `policy::AllOf` stayed the same, as in representing a conjunction for the target definition, the transformation is easily possible. The differences to our metamodel are only some left-out elements. Equation 4.1 illustrates the transformation rules. XACML expects a target definition for every PolicySet, Policy and Rule. Our model encapsulates this in the `policy::AllOf`. If there was no `policy::AllOf` element, an empty `xacml::Target` is created, such as at line 3 or 6 in Listing 4.2. If there exists at least one `policy::AllOf` like in line 24 from Listing 4.1, for each `policy::Allof` a `xacml::AnyOf` is created. The `xacml::AnyOf` then has one `xacml::AllOf`, which contains the `Match` elements from our model. Every `policy::Match` is transformed to `xacml::Match` elements. These can be multiple elements since not every `policy::Match` can be reduced to one `xacml::Match` (see MatchTransformation). For our target definition in line 24 from Listing 4.1, the transformation results look like the target starting from line 9 from Listing 4.2.

The next custom elements are the match elements. We provide four matches `EntityMatch`, `MethodMatch`, `GenericMatch`, `XMLMatch`. The transformation for the first three elements is illustrated in Figure 4.8. For each match, the transformation is separated by a horizontal line. The common ground for

the three matches is that they are contained by an `AllOf` element. In addition, the transformation always creates `xacml::Match` elements with corresponding `xacml::AttributeValue` elements for the reference values and `xacml::AttributeDesignator` for the selection of the attributes. As an alternative, XACML would also allow using a `AttributeSelector` in a match. However, the `AttributeSelector` is optional and, therefore might not be available in all implementations. In addition, the usage might be more complicated since it uses XPath expressions [207] to identify the attributes. Therefore, we choose the variant with the `AttributeDesignator`. We repeated this design decision also in other transformations, where there is a choice between `AttributeDesignator` and `AttributeSelector`. We will now explain each transformation in more detail.

The `policy::EntityMatch` is used to identify an architectural element, such as a `ResourceContainer` or `AssemblyContext`. The simplified identification of these elements happens by using the id and the name of the architectural element. Each element is an `Entity` in PCM and therefore needs to have these attributes. The id and name are from the type `string`. Hence, the datatypes in the XACML elements are strings. The `xacml::AtributeValue` sets the reference value, and the `xacml::AttributeDesignator` selects the values from the request. The attribute is selected based on the `category` from the `policy::EntityMatch`. The `Category` in `xacml::AtributeDesignator` is directly taken from the `policy::Entitymatch`. The `AttributeID` is the name for the category extended with *"id"*. For instance, for the category resource, the id would look like *"resource:resource-id"*. Because the reference value with the `xacml::AttributeValue` and the attribute selection from the request with the `xacml::AtributeDesignator` both have the datatype string, we use the `string-equal` as the compare functions. It compares two strings for equality and is set by the `MatchID` in `xacml::Match`.

The `policy::MethodMatch` transformation is separated in two cases depending on the concrete subclass of the contained `policy::MethodSpecification`. For both cases, the transformation results into two `xacml::Match` elements. The first is to identify the instance of the service, and the second is to identify the service. Splitting the selection into two matches is not problematic since the conjunction of the matches guarantees that both matches always need to be fulfilled for the target selection. The transformation with the `policy::ServiceSpecification` consists of two parts. The first part is to identify the instantiated component where the service is running, and the second part is for the identification of the service. The first `xaml::Match` is

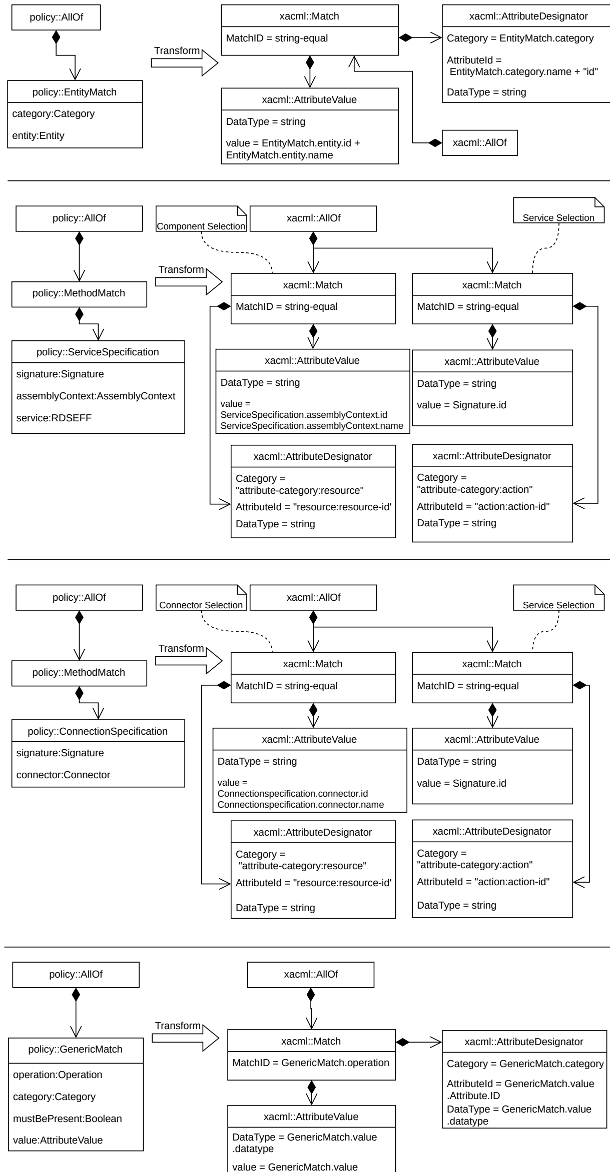


Figure 4.8.: EntityMatch, MethodMatch and GenericMatch transformation

for the component selection and is similar to the transformation with the `poliy::EntityMatch` with the exception that the `entity` is replaced by its child element `AssemblyContext` representing only an instantiated component and the direct setting of the resource category. The second `xacml::Match` is used for the identification of the service. For the service, we choose the string datatype again. Therefore, the `MatchID` is again `string-equal`. In our case, we choose to identify a service by its `id`. Therefore, we use the `Signature.id` as the value for the reference value. XACML provides a special category for actions performed on the system. In our case, we see a service also as an action. In our transformation, the service is also stored in the action category. Hence, the `Category` in the `xacml::AtrributeDesignator` is selecting the action category by using *"attribute-category:action"*. The `AttributeId` is then the suggested attribute used by XACML to identify an action. An example of this transformation can be found in Listing 4.2 starting from line 12. The first match elements describe the service selection, and the second match (starting from line 16) the component selection.

In the case with a `policy::ConnectionSpecification`, the `xacml::Match` for the service selection is the same as in the case with the `policy::Service-Specification`. However, in the `xacml::AttributeValue` contained by the other `xacml::Match`, the value is replaced by the id and name of the `connector` from the `policy::ConnectionSpecification`. The other elements stay the same.

For the `policy::GenericMatch` transformation, the `MatchID` is determined by the selected `operation` from `policy::GenericMatch`. The `xacml::Attribute-Designator.category` is also directly taken from the original element. The datatypes, the actual reference value, and the `AttributeId` are derived from the `policy::GenericMatch.value`

The last match transformation is the `XMLMatch`. Here, software architects already specify a match in XACML. Therefore, we do not need to transform it to an XACML statement. Yet, we still need to add it at the right position in the XACML model. One possible solution could be that we just add the textual description in the output XACML file. However, in this case, we would need to somehow store the information where to add the `XACML` strings and also consider the correct syntax for adding an element. In addition, the internal XACML model would be incomplete, and only the serialized XML file would be a complete model. This is especially for referenced values not good. Therefore, we choose to first parse the XACML statement with the Java

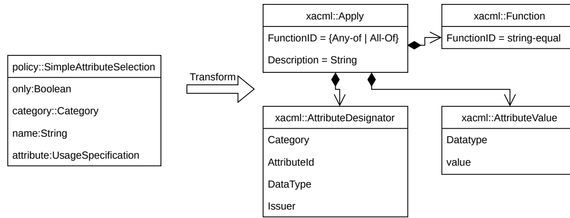


Figure 4.9.: SimpleAttributeSelection transformation

Marshaller, which results in a XACML model of the statement. This model is then added to the internal XACML model during the transformation.

The transformations for `Policy`, `Rule`, `VariableDefinitions`, `VariableReferences`, `FunctionReferences`, `AttributeDesignator`, and `Apply` are straightforward. It is copying the values from our policy metamodel to the matching value in XACML.

The custom element `XMLString` is transformed in the same way as `XMLMatch` by using the Java Marshaller.

The `policy::SimpleAttributeSelection` basically encapsulates the condition to check for an attribute. The transformation creates the XACML elements along the description of the bag comparison in XACML [132, A.3.12]. The basic idea is to use an `xacml::Apply`, and its parameters are a `xacml::Function`, a `xacml::AttributeValue` and a `xacml::AtributeDesignator` to select the bag from the request attributes.

The attributes from `xacml::Apply` are derived from the following. For the `FunctionID`, depending on the `only` value, either `Any-Of` or `All-Of` is chosen. The difference is that for `Any-Of`, there could be different values in the bag derived from the `xacml::AttributeDesignator`. In the case of `All-Of`, all values need to be the reference value. An example of this behaviour is that a requestor has multiple roles. Depending on the `only` value, it is allowed to have multiple roles for a requestor, or it is forbidden. The `Description` is derived from the `name`. The compare function is set within the `xacml::Function` with the value `FunctionID` to `string-equal`. This compares the bag values to the reference value (`xacml::AttributeValue`). The `xacml::AttributeValue.Datatype` and `xacml::AttributeValue.value` is derived from the `policy::SimpleAttributeSelection.attribute`. The third

parameter is the bag selection by the `xacml::AttributeDesignator`. The `Category` can be directly derived from the `category` from the original element. The `AttributeId` and `Datatype` can be derived as with the `xacml::AttributeValue` from the `attribute`. The `Issuer` is an optional attribute. It indicates the origin of an attribute. In our case, we set it if the `attribute` contains an `EntityAttribute`. Then it is set to the id of the architectural element issuing/producing the attribute.

An example of this transformation can be found in Listing 4.2 with the apply element starting from line 26. Starting from line 32, it shows an example where an issuer is used. The issuer is the id of the entity which generates the attribute. In our case, it is the `AssemblyContext` for the `MachineComponent`. We can see that the same id is used during the match in line 17.

Besides the policy transformation, the second transformation type is the access request transformation. The transformation is exemplarily illustrated in the lower parts at Figure 4.5. At the start, the PDP loads our transformed XACML model. The analysis then creates based on the software architecture and the policy model a request. The request is already structured according to a XACML request. Hence, it assigns attributes to the different categories. However, the attributes still use our policy metamodel. In the next step, the request is transformed to a valid XACML request. An example is shown in Listing 4.3. Then the PDP can evaluate the request, and based on the result, the analysis builds the PDRResult model. The metamodel for this is illustrated in Figure 4.10. This structure is useful since it decouples the request from the request evaluation. It enables us to replace the used PDP with any other XACML compatible PDP. We will now explain the necessary steps in more detail.

During the request creation, we assign the requestor's context and the requested architectural element to the categories subject (for the requestor), environment, resource (for the requested element) and action. Some of these attributes are already preassigned during the attribute creation, and others are dynamically derived, such as the resource. Afterwards, we have a list of `UsageSpecifications` for each category.

The next step is to transform the list to an XACML request similar to the request in Listing 4.3. For this, we transfer each attribute, similar to the policy transformation for an XACML element. Listing 4.3 illustrates the request for the technician to access the machine. The subject attributes are listed starting from line 2. In our case, this is the role with `Technician`. The resource

Listing 4.2: Simplified XACML file for the technician scenario from the running example

```xml
<PolicySet xmlns="core:schema:wd-17" PolicySetId="_d7xFEZSjEeyvBd3n0aDE-g" Version="0.0.1"
      ↳ PolicyCombiningAlgId="policy-combining-algorithm:deny-unless-permit">
    <Description>Policies for MaintenanceSystem</Description>
    <Target/>
    <Policy PolicyId="__-SiIbQGEeyBBMZUdAqcvg" Version="0.0.1"
          ↳ RuleCombiningAlgId="rule-combining-algorithm:deny-unless-permit">
        <Description>Machine</Description>
        <Target/>
        <Rule RuleId="_BcIZILQHEeyBBMZUdAqcvg" Effect="Permit">
            <Description>Technican Access</Description>
            <Target>
                <AnyOf>
                    <AllOf>
                        <Match MatchId="function:string-equal">
                            <AttributeValue DataType="string">_HZ11cJSREey0ldzBRFqWVw
                                  ↳ </AttributeValue>
                            <AttributeDesignator Category="attribute-category:action"
                                  ↳ AttributeId="action:action-id" DataType="string"/>
                        </Match>
                        <Match MatchId="function:string-equal">
                            <AttributeValue DataType="string">_4lwccJSYEeyjlrv9ryW3Zw
                                  ↳ Assembly_MachineComponent</AttributeValue>
                            <AttributeDesignator Category="attribute-category:resource"
                                  ↳ AttributeId="resource:resource-id" DataType="string"/>
                        </Match>
                    </AllOf>
                </AnyOf>
            </Target>
            <Condition>
                <Apply FunctionId="function:and">
                    <Description>aName</Description>
                    <Apply FunctionId="function:any-of">
                        <Description>aName</Description>
                        <Function FunctionId="function:string-equal"/>
                        <AttributeValue DataType="string">Technican</AttributeValue>
                        <AttributeDesignator Category="subject"
                              ↳ AttributeId="_f4Ph0rGMEeyLRdjos_zQbg" DataType="string"/>
                    </Apply>
                    <Apply FunctionId="function:any-of">
                        <Description>aName</Description>
                        <Function FunctionId="function:string-equal"/>
                        <AttributeValue DataType="string">Failure</AttributeValue>
                        <AttributeDesignator Category="attribute-category:resource"
                              ↳ AttributeId="_S5hooLGMEeyLRdjos_zQbg" DataType="string"
                              ↳ Issuer="_4lwccJSYEeyjlrv9ryW3Zw"/>
                    </Apply>
                </Apply>
            </Condition>
        </Rule>
    </Policy>
</PolicySet>
```

Listing 4.3: Generated XACML request for the technician scenario

```
1  <Request ReturnPolicyIdList="true" CombinedDecision="false" xmlns="core:schema:wd-17">
2      <Attributes Category="subject">
3          <Attribute AttributeId="_f4Ph0rGMEeyLRdjos_zQbg" IncludeInResult="false">
4              <AttributeValue DataType="string">Technican</AttributeValue>
5          </Attribute>
6      </Attributes>
7      <Attributes Category="attribute-category:environment"/>
8      <Attributes Category="attribute-category:resource">
9          <Attribute AttributeId="_S5hooLGMEeyLRdjos_zQbg" Issuer="_4lwccJSYEeyjlrv9ryW3Zw"
               ↳ IncludeInResult="false">
10             <AttributeValue DataType="string">Failure</AttributeValue>
11         </Attribute>
12         <Attribute AttributeId="resource:resource-id" IncludeInResult="false">
13             <AttributeValue DataType="string">_4lwccJSYEeyjlrv9ryW3Zw
                   ↳ Assembly_MachineComponent</AttributeValue>
14         </Attribute>
15     </Attributes>
16     <Attributes Category="attribute-category:action">
17         <Attribute AttributeId="action:action-id" IncludeInResult="false">
18             <AttributeValue DataType="string">_HZ11cJSREey0ldzBRFqWVw</AttributeValue>
19         </Attribute>
20     </Attributes>
21 </Request>
```

attributes are listed starting from line 8. The first attribute is the machine states with `Failure`, which resonates from the `Machine`. Hence, it contains the `Issuer` with the `Machine`'s `ID`. The second attribute is the `Machine`. The third attribute category is the action, starting from line 16. It contains the `ID` of the called service. The PDP then uses the described policy in Listing 4.2, matches the request on it, and then evaluates it. In our case, since the `Machine` and `Service` matches the target for the Rule ("`_BcIZILQHEeyBBMZUdAqcvg`"), the condition is evaluated. The conditions evaluate to `Permit` since the role and machine state match the expected values.

This decision of the PDP must be then transferred to the analyses. To transfer the result, we develop a simple data exchange metamodel, illustrated in Figure 4.10. The `PDPResult` encapsulates the access decision with the `decision`. A decision can be analogues to XACML either `PERMIT`, `DENY`, `INDETERMINATE`, `NOT_APPLICABLE`. `PERMIT` and `DENY` directly follow from the rule effect. `INDETERMINATE` is, when the PDP cannot decide a concrete rule. This can be, for instance, if an error happens or there are multiple contradicting effects. `NOT_APPLICABLE` indicates that no rule could be applied for the request from the PDP. In addition to the `decision`, the `PDPResult` stores also with the `policyIdentifiers` a `String` list with the identifiers involved in

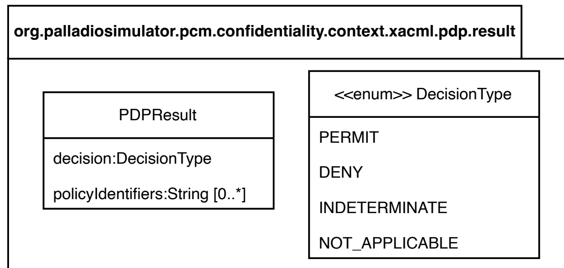


Figure 4.10.: Result model for PDP decisions

the policy decision. For instance, with our example request (Listing 4.3) and policy (Listing 4.2) this is "`__-SiIbQGEeyBBMZUdAqcvg`". These identifiers are not useful for the analysis itself. However, they help to interpret the results of an analysis since they enable the tracing of a decision back to the involved policies. In addition, they are helpful during the debugging of the analyses.

4.3. Considering Vulnerabilities in Software Architecture Models

This section describes our modelling concept for vulnerabilities in software architecture. Together with the metamodels for attacks (c.f. Section 4.4), and attackers (c.f. Section 4.5), it forms our contribution C2. The contribution C2 answers together with contribution C1 our research question **RQ2.1** and both build the foundation to answer the research question **RQ2.2**. We introduced the vulnerability metamodel in Walter et al. [211]. Afterwards, we extended the metamodel in Walter et al. [212] for further security properties and in Kirschner et al. [94] we extended the metamodel to support the automatic extraction of security properties.

Besides the access control policies, other properties influence the security of a system. For instance, the vulnerabilities of a system can be used to compromise a system and affect its overall security. Vulnerabilities can be found in different architectural elements, such as the components or deployed network resources. Usually, a system can contain known vulnerabilities, such as Log4Shell [130]. Log4Shell can enable attackers to exploit arbitrary code

and thereby enable them to compromise an architectural element. In addition to the known vulnerabilities, a system contains unknown vulnerabilities. In the case of unknown vulnerabilities, security experts often determine potential security threats, for instance, with threat models. However, a vulnerability alone is not enough to determine the security influence since the exploitation often depends on the context and properties of a vulnerability. For instance, the access control approach can prohibit the exploitation or enable an exploit. Therefore, it is necessary to model vulnerabilities together with their properties. These models can then be used during the analyses to decide whether they can be exploited. For specifying models with a well-defined syntax, we developed a metamodel and integrated the metamodel in PCM.

In this section, we will first describe in Section 4.3.1 the requirements, which guided the development of the metamodel. Afterwards, we describe our vulnerability identifiers in Section 4.3.2. In Section 4.3.3, we describe our vulnerability metamodel and the integration into PCM in Section 4.3.4. Section 4.3.5 describes an approach to automatically derive vulnerabilities.

4.3.1. Requirements for Modelling Vulnerabilities

During our work, we identified different additional requirements besides the purpose to express vulnerabilities, the exploitation of vulnerabilities and attackers. These additional requirements guided the metamodel development and helped us to identify important properties. We derived the requirements iteratively based on our work in different research projects, related work and vulnerabilities databases. As a starting point, we used our initial case studies [11, 169, 12] and existing vulnerability description from databases, such as the NVD [131] or CAPEC [38]. We iteratively developed the metamodel and redefined the requirements based on found use cases and scenarios. In the following, we will first list our identified requirements and afterwards discuss the concrete reasoning. Our final set of requirements are:

VR1 Security experts should be able to define vulnerabilities for software architecture layers.

VR2 Security experts should be able to define concrete and non-concrete vulnerabilities.

VR3 Security experts should be able to reuse knowledge about existing concepts for the definition of vulnerabilities.

The first requirement **VR1** describes similar to **AR1** that we need to annotate vulnerabilities on different layers in the software architecture. This is necessary since vulnerabilities can be found on different levels of abstraction. We started the metamodel to describe vulnerabilities only for the components of a system. This is very common and are also performed by other approaches, such as CVE. For instance, the Log4Shell [130] vulnerability is a software vulnerability. It can be annotated in ADLs to existing components and then represents a vulnerable component. Yet, ADLs like PCM differ between instantiated components (`AssemblyContext` in PCM) and component types (e.g. `BasicComponent` in PCM). Also, for a vulnerability, a differentiation is useful since some vulnerabilities only apply to certain configurations. Here, the instantiated components can be used to differentiate between different configurations, which might be vulnerable or not. Therefore, it is important to differ between the type and the instance level and the modelling approach should consider these differences. However, during our work, we also identified the need to consider other architectural layers, such as the network resources and the hardware resources. For instance, vulnerabilities, such as Meltdown [105] or Spectre [95] are hardware vulnerabilities. Therefore, it is necessary to consider vulnerabilities also in other architectural elements besides the software components. In addition, there can be different abstraction layers for a software architecture. For instance, the operating system can be modelled like in our running example as a `ResourceContainer` and not as a separate component.

VR2 describes that the vulnerability model contains concrete vulnerabilities like the Log4Shell vulnerability. Concrete vulnerabilities are known and described in existing databases, such as the NVD. Different studies, such as a study by Unit 42[7] or a report by the UK government [84] show that software contains known vulnerabilities or outdated software, which again usually contains known vulnerabilities. These studies show the need to consider known vulnerabilities because they are a real problem and exist. Therefore, we added the first part regarding the known vulnerabilities to

[7] J. Greig. *96% of third-party container applications deployed in cloud infrastructure contain known vulnerabilities: Unit 42.* en. Oct. 2021. URL: `https://www.zdnet.com/article/96-of-third-party-container-applications-deployed-in-cloud-infrastructure-contain-known-vulnerabilities-unit-42/` (visited on 04/03/2023).

our requirement **VR2**. We derived the second part of the requirement based on the fact, that not all attacks are known. Either because we are still in an early development phase, where concrete aspects are unknown, for instance, if no implementation exists or the concrete hardware is not yet chosen. In other cases, it can be that so far no vulnerabilities have been identified in the system. In these cases, security experts often perform a threat analysis for the system and identify possible threats in the system. These threats are often not concrete in the sense that they describe concrete exploits. They describe an abstract concept of how to compromise an architectural element. For instance, the CWE concept enumerates multiple possible weaknesses, such as no input validation [113] or the usage of hard-coded credentials [116]. However, considering these threats as vulnerabilities is beneficial because it enables security experts to analyse the system and give an early estimation of potential security incidents. For us, these threats represent non-concrete vulnerabilities. Therefore, we added them as part of the requirement **VR2**.

The last requirement **VR3** describes the reuse of knowledge and concepts by security experts. There exist databases which document security vulnerabilities, such as the NVD or various private company sites like, e.g., for Ubuntu[8] or the Microsoft Security Response Center[9]. These services provide similar classifications for vulnerabilities and experts usually are familiar with the classifications. Using similar concepts can help an expert since it saves effort to learn a completely new concept. In our case, we see concepts as similar if they contain the same attributes and these attributes are derived similarly. For instance, a commonly used concept of vulnerability databases is CVSS. This concept contains the attribute *attack vector* with the meaning that it classifies the location from which a vulnerability can be exploited. Based on our requirement, we expect that if the metamodel also has an attribute attack vector, the semantics would be the same or similar. Another advantage of having a similar concept is that the knowledge about classified vulnerabilities can be reused. The classification of a new vulnerability is hard and many different aspects like the complexity or the impact need to be considered. One example for the complexity of the classification is the vulnerability in our running example (c.f. Chapter 3). During our work on this thesis, the used vulnerability CVE-2021-28374 was reclassified [129]. This shows that

[8] *CVE reports*. URL: `https://ubuntu.com/security/cves` (visited on 01/17/2023).

[9] *Security Update Guide*. URL: `https://msrc.microsoft.com/update-guide/vulnerability` (visited on 01/17/2023).

for some vulnerabilities, the classification is not easy. Hence, reusing the knowledge of the databases can reduce the effort to classify complicated vulnerabilities. However, for reusing these values, our metamodel needs to use similar values. Therefore, we choose **VR3** as a requirement.

4.3.2. Modelling Identifiers for Vulnerabilities

During the modelling and the analysis, we need to identify a vulnerability. One possibility to identify a vulnerability is to reuse existing identifiers or classifications. In our case, we choose to reuse the concepts CWE and CVE. These are commonly used concepts to identify weaknesses and vulnerabilities. We use CVEs to identify concrete vulnerabilities. For many known vulnerabilities CVEs exist. For the non-concrete vulnerabilities, we use CWEs. These are used to categorize common security weaknesses in hardware and software.

We integrated these concepts into our metamodel. Figure 4.11 illustrates the elements involved in the identification. The common parent element is the abstract `AttackCategory`. A CWE is encapsulated by the `CWEID`. The actual category is stored in the `cweID` as a String. An example value can be *"CWE-312"* which represents the "Cleartext Storage of Sensitive Information" [114]. CWE also is hierarchical with multiple parents and children. For instance, CWE-312 has, among others, CWE 922 with "Insecure Storage of Sensitive Information" [117] as a parent. It also has multiple children, such as CWE-313 or CWE-314. This relationship to other CWEs can be set by the `parents` and `children` attributes.

The `CVEID` encapsulates a CVE identifier. The value is stored in the `cveID` as a string, similar to the CWE. For example, the vulnerability in our running example would be stored as "*CVE-2021-28374*" [129]. A CVE also has often relating CWEs. In our case, we choose to model this not in the identifiers, but it is encoded in later introduced elements.

4.3.3. Modelling Vulnerabilities

For modelling different vulnerabilities in the architectural model, we developed a vulnerability metamodel. The key idea of the metamodel is to reuse existing knowledge about vulnerabilities and their classifications. Therefore,

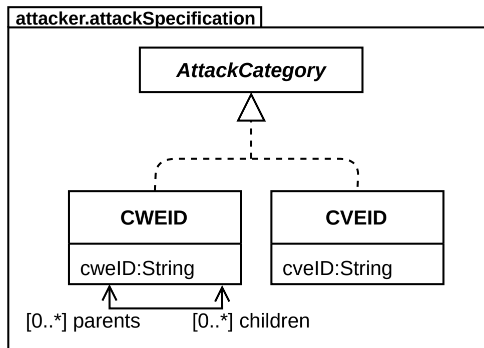


Figure 4.11.: Category metamodel elements based on [211]

we analysed different existing classifications and descriptions, such as CWE, CVE and CVSS.

Figure 4.12 illustrates an excerpt of the vulnerability metamodel. The grey elements are existing elements from PCM. The metamodel contains two types of vulnerabilities: `CVEVulnerability` and the `CWEVulnerability`. The first is built on the concept of CVEs and the second is based on CWEs. They share the same parent classes. One parent class is the abstract `CWEBased-Vulnerability`. It encapsulates the reference to the CWEs with its `id` attribute. Both vulnerabilities share this since a CVE matches to CWE-categories. For instance, the vulnerability "*CVE-2021-28374*" in our running example has the CWE *"CWE-312"*. In some cases, there can be more than one CWE for a CVE. Hence, the `id` can have multiple `CVEIDs`.

Remark: During the writing of the thesis, the CWE class from the vulnerability in the running example was remapped to CWE-712 [129]. In our thesis, we continue to use the old classification.

The impact and further properties of a vulnerability are described in the abstract `Vulnerability`. It is also a parent element for the `CVEVulnerability` and the `CWEVulnerability`. We differ in the attributes similar to CVSS between attributes describing the exploitability and attributes describing the impact [44]. Both attributes are either based on the description of relevant attributes from vulnerability databases and security incidents or are based on the *Base Metric Group* from the CVSS calculation [44]. However, we do

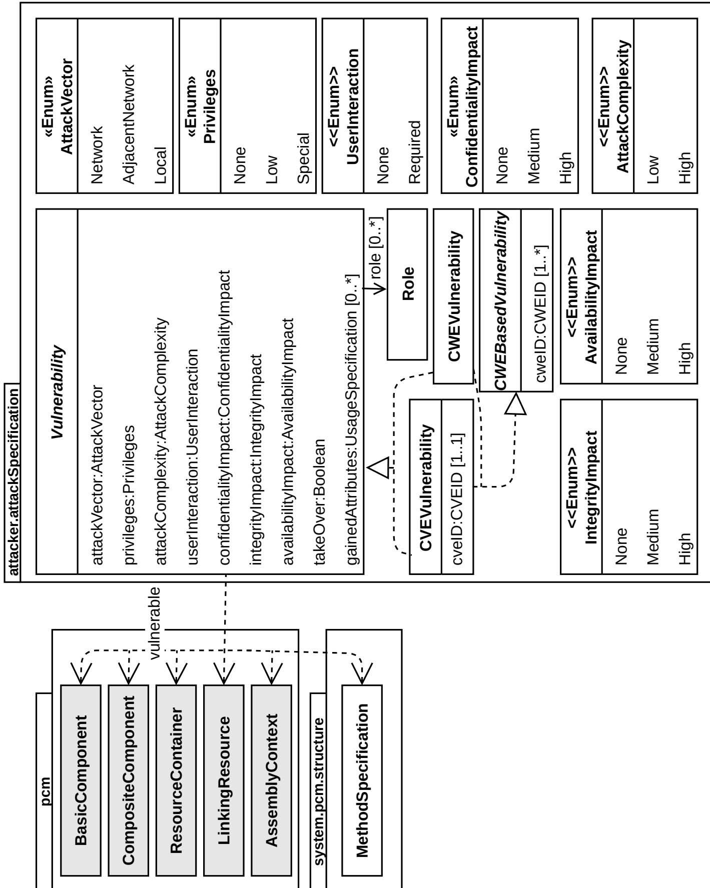


Figure 4.12.: Vulnerability metamodel elements based on [211]

not use from CVSS the scoring. We only use the metrics used for the scoring calculation. It is also not necessary to use CVSS to gather the necessary values. Other similar approaches, such as Common Weakness Scoring System (CWSS) [112], can be used as long they share similar properties. Reusing these values is beneficial since determining the metrics and classifying a vulnerability is complicated and with the reuse, we can reduce the effort of manually determining the attributes. In addition, using a possible familiar description fulfils **VR3**. To get a better understanding, which elements are derived from CVSS, we mark them with an asterisk (*) at their initial description. We will start first the description with the exploitability attributes and afterwards describe the impact:

- `attackVector*`: The attribute uses the enumeration `AttackVector` to describe from which locations a vulnerability can be exploited. The location is structured similar as in CVSS with `Network`, `AdjacentNetwork` and `Local`. `Network` indicates that the vulnerability can be exploited from any architectural element in the network, which can reach the vulnerable element. The attacker does not need to be in the same subnetwork. The `AdjacentNetwork` indicates that the vulnerability can only be exploited from the same network. In CVSS, this can be a logical or physical network [44]. In our case, this is problematic since PCM does not differ explicitly between different network zones logically and also, the physical network is very abstract. We solved this by assuming that, for us, an adjacent network is a network between a `LinkingResource` and its connected `ResourceContainers`. `Local` represents that the vulnerability cannot be exploited over the network but only locally. In our case, this is the deployment relationship between an `AssemblyContext` and a `ResourceContainer`. A locally vulnerable `ResourceContainer` can be compromised from an `AssemblyContext` deployed on it.
- `privileges*`: This describes whether, for the exploitation of the vulnerability, certain privileges are necessary. The privileges are described like in CVSS in the enumeration `Privileges`. It differs between `None`, `Low`, and `Special`. `None` states that the vulnerability does not need any authorisation or privileges to be exploited. Therefore, anyone can exploit it. `Low` states the vulnerability can be exploited only by authorised users and `Special` states that special authorisation is necessary, such as certain administrative privileges. We choose to support these three categories in the metamodel so that it is more similar to the original

CVSS metric. However, during the analyses, we only differ between authorised and non-authorised. Hence, `Low` and `Special` are considered as one element since PCM currently does not support different authorisation levels.

- `attackComplexity*`: The complexity of exploiting a vulnerability can vary. Some vulnerabilities like the Log4Shell are rather easy to exploit [130]. Others are more complex. Understanding the complexity of the used vulnerability can give an expert the first indication of how critical the vulnerability is. We also use it in Walter et al. [209] to give an estimation of the criticality of attack paths. In our metamodel, we differ in the enumeration `AttackComplexity` similar to CVSS between `Low` and `High`. `Low` indicates an low attack complexity and `High` a high.
- `userInteraction*`: The attribute `userInteraction` describes as in CVSS, that for the exploitation another user is necessary. This can be a second malicious user or a regular user, who unintentionally helps a malicious user to exploit a vulnerability. We differ here between `None` and `Required`. These elements are similar to CVSS. In our analyses, we do not consider additional user interaction for the vulnerabilities. However, we support the attribute with filter operation.
- `role`: We derived this element from vulnerability descriptions in databases. Some applications are only vulnerable if another application with a certain role is attacking them. For instance, clients like the Nextcloud client can be vulnerable to a malicious server [128]. In our case, we model this scenario by adding the role *Server*. This then requires that the vulnerability can only be exploited from an entity with the role *Server*. As a remark, the role is here independent and not to be confused with user roles or access control roles from the Section 4.2.

Besides the attributes describing the vulnerability's exploitability, the metamodel also contains the impact of a vulnerability. The impact of confidentiality can differ. For instance, the vulnerability in our running example can lead to the loss of credentials. Other vulnerabilities might not have the same result. Our impact attributes are:

- `confidentialityImpact*`: Based on the CVSS metric, the enumeration `ConfidentialityImpact` describes how the vulnerability affects the confidentiality of the architectural element. The metamodel differs as

in CVSS between None, Medium, and High. In our case, we use these values to indicate whether data from an architectural element can be stolen by exploiting the vulnerability. This assumes that all data is considered to some extent confidential. As for the extraction, we do not differ between Medium or High. Both elements lead to an extraction. This is based on the fact that the PCM currently does not differ between confidentiality levels. However, in the future, it might be possible. Therefore, we keep the differentiation and in addition, our modelling sticks closer to the existing standard metric from CVSS.

- integrityImpact*: Similar to the CVSS metric, the enumeration IntegrityImpact describes whether exploiting a vulnerability can affect the integrity of the architectural element and its managed data. An attacker might not see the data as with the confidentialityImpact, but the data could be changed or modified. We differ here based on the CVSS metric between None, Medium, and High. None indicates no impact, Medium indicates an impact on some parts and High indicates that nearly everything is affected. In our analyses, we do not support this attribute besides in filtering paths. However, we included the attribute so that experts could see this in the result and for future extensions of the analyses.
- availabilityImpact*: Similar to the previous impact attribute the AvailablityImpact describes what affect the vulnerability has on the availability. The values are extracted from CVSS and we differ again between None, Medium, and High. None stands for no impact, Medium if only smaller parts are affected, and High if nearly all parts or the critical parts are affected. In this work, we focus on confidentiality. Therefore, our analyses only support the availability impact by using it as a filter attribute.
- takeOver: This attribute describes with a boolean whether, by exploiting the vulnerability, an attacker could gain full control of the element. This attribute stems not from CVSS, but it usually can be extracted from vulnerability descriptions, such as in databases like the NVD. We need this attribute because the previous impact description can mean that the attacker has full control. However, it is not necessarily clear. For instance, a vulnerable database component with a High confidentiality impact could just leak the data and not give the attacker full access. On the other hand, even a fully controlled hardware resource

does not automatically result in a confidentiality impact since data could be encrypted like with end-to-end encryption [121].

- `gainedAttributes`: This attribute describes that, by exploiting a vulnerability, the attacker could gain the listed attributes. This attribute is derived from public descriptions of security incidents and databases. For instance, the vulnerability in our running example states in the description that it can leak credentials [129]. Based on this description, an expert can then assign with our metamodel which credentials they are. In our case, we model credentials as a list of `UsageSpecifications`. In this way, it is consistent with the access control metamodel (c.f. Section 4.2).

Many of these attributes are also system independent. This means that experts only need to create the model once and the model can be transferred to other software architectures. In general, all the CVSS based attributes are system independent. In addition, the `role` and `takeOver` is system independent. Nevertheless, there could be certain instances where it is not possible. This is similar to the CVSS specifications, which allows for customisation since sometimes a vulnerability is dependent on the concrete configuration. However, in these cases, the experts can adapt the model to their liking. In contrast, the `gainedAttributes` is system dependent since it contains the concrete credentials for a system. Usually, the access control model is specific to the concrete system. Nevertheless, in some cases, it is possible to transfer the access control model or at least the role/attribute model. In these cases, the attribute is considered system independent.

A `Vulnerability` is connected to different architectural elements. The vulnerability metamodel supports the annotation of a `BasicComponent`, `CompositeComponent`, `ResourceContainer`, `LinkingResource`, `AssemblyContext`, and `MethodSpecification`. In Figure 4.12, we marked the annotation with dashed lines with open arrows and the name vulnerable. We use this special syntax to indicate that the annotation is more complex and will be explained later on. However, for simplicity reasons, we want to show here the architectural elements. The different elements cover different layers with `ResourceContainers` as hardware layers or `AssemblyContext` for logical ones. Hence, it covers also **VR1**. So far, not every analysis supports every architectural element. Especially, the `CompositeComponent` is not supported by any attack analyses. However, for completeness, we add it to the vulnerability model. Future versions of the attack analyses may support it.

Listing 4.4: Simplified textual representation of the vulnerability model instance from the running example

```
CVEVulnerability {
    cveID CVEID{
        cveID: "CVE−2021−28374"
    },
    cweID CWEID{
        cweID: "312"
    },
    attackVector: Network,
    privileges: None,
    attackComplexity: Low,
    userInteraction: None,
    confidentialityImpact: High,
    integrityImpact: None,
    availablityImpact: None,
    takeOver: true,
    gainedAttributes: {admin},
}
```

An example model instance is illustrated in Listing 4.4. It is a textual representation of the vulnerability “CVE-2021-2874” from our running example. We first have the CVE and corresponding CWE category. These values can be directly read from the NVD homepage. Then the attack vector, privileges, user interaction, attack complexity, confidentiality impact, integrity impact and availability impact are specified. These values can be directly extracted from the CVSS vector. The vector for our vulnerability looks like “CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:N/A:N” [129]. The vector consist of pairs with types and values. Each pair is separated by a slash (“/”). The first pair defines the used CVSS version. Afterwards, the different base metrics are described. For instance, *AV:N* stands for *Attack Vector: Network* and states that the attack vector is network. Based on the vector, these properties can be filled. The last two elements in our model instance are filled based on the description of the vulnerability and on the individual modelled system. In our case, this is that the component can be compromised by the vulnerability and that the vulnerability can leak the admin attribute.

4.3.4. Integration in Palladio

Up to now, we cannot annotate existing PCM elements with our vulnerabilities. In Figure 4.12, the annotation is only hinted by the dashed line, but the concrete realisation is not given. However, the annotation is necessary because, without it, it is unclear which vulnerability belongs to which architectural element. Therefore, it is necessary to integrate the vulnerabilities into the PCM.

The main concept for the PCM integration is that we want to extend existing architectural elements with the `Vulnerability` property. Strittmatter [192] describes this behaviour as a *Metamodel Extension.* He proposed in his work different mechanisms to realize a model extension. In our case, we chose the method "Referencing with External Container" [192, p. 109]. This decision is mainly based on technical reasons. The inheritance approach would be technically feasible as well. However, the editor creation would be much more cumbersome because of the internal PCM configuration. The other extension mechanism over profiles was technical not reliable. The referencing approach works on a technical level reliable and adds the benefit that the vulnerability package itself is independent of the used ADL.

Figure 4.13 illustrates the integration into PCM. The integration is contained within a separate metamodel package to separate the PCM-specific elements and the non-PCM-specific elements. The main concept is that we form a link between our new metamodel elements and the architectural elements. The abstract class `SystemIntegration` stores the link to our new metamodel elements. It contains one `PCMElement`. `PCMElement` reflects the PCM integration. It inherits from `RepositoryElement`, `ResourceEnvironmentElement`, `SystemComponent` (indirectly), and `SystemElement` the association to architectural elements. Therefore, the metamodel supports the extension of the following architectural elements: `CompositeComponent`, `BasicComponent`, `ResourceContainer`, `LinkingResource`, `AssemblyContext`, and `MethodSpecification`.

The extension or annotation for the architectural elements is specified by the subclasses from the `SystemIntegration`. This element links the architectural element with the annotation. There are three concrete subclasses:

- `VulnerabilitySystemIntegration` references a `Vulnerability` and links the selected archiectural element to the selected `Vulnerability`.

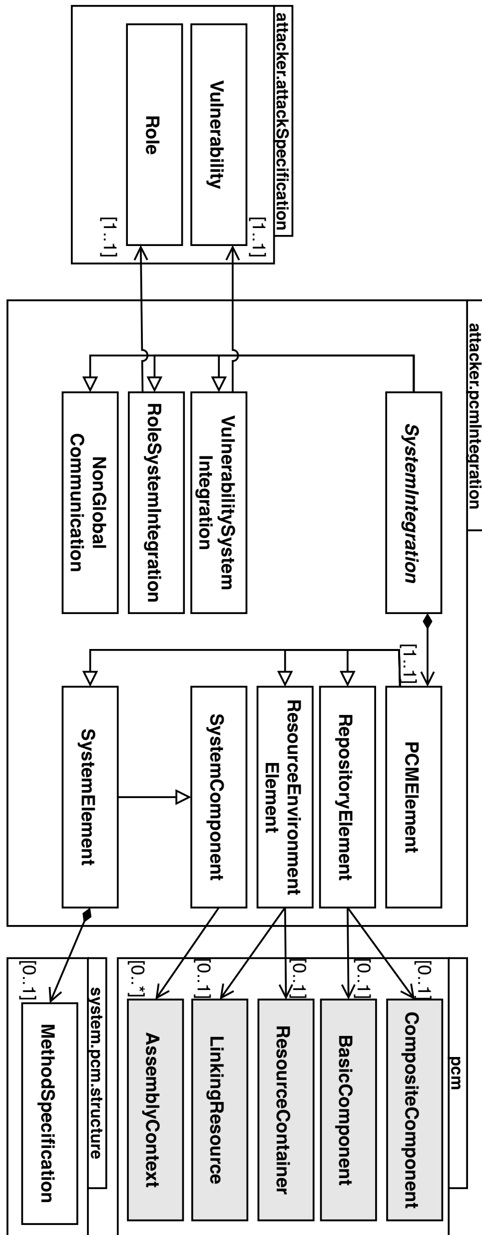


Figure 4.13.: Vulnerability PCM integration

- `RoleSystemIntegration` references a `Role`. Hence, it assigns the `Role`, such as Server, to an architectural element.
- `NonGlobalCommunication` indicates a special element which cannot communicate globally. In PCM, there are two layers for communication. It is the logical communication with `AssemblyConnectors` between components and the hardware layer between a `LinkingResource` and `ResourceContainers`. There can be different ways how to interpret the logical connection in regard to a malicious user. It might be that the components can only communicate over logical connections and this is enforced by settings or other network mechanisms, such as a firewall. However, it could also be that every other component in the network could access it. Since this behaviour is not completely clear, we introduce a boolean flag for components indicating whether it is the first behaviour or the second.

4.3.5. Automatic Derivation of Vulnerabilities

Remark: In the previous sections and in the sections afterwards, I was the lead contributor. In this subsection, we discuss a shared contribution. For the extension of the attack propagation analysis and the corresponding metamodel extension to support the automatic derivation, I was the lead contributor. The shared contribution is the identification of the vulnerabilities and the automatic vulnerability model creation based on our vulnerability metamodel. Figure 4.14 illustrates this by using the grey background for artefacts where I am the lead contributor and black for artefacts where I was the co-contributor.

Since the vulnerability metamodel elements are based on commonly used approaches and some of these attributes are publicly available in databases, it is possible to automate the derivation of some vulnerabilities.

In a publication [94], which I co-authored, we investigated the automatic derivation of vulnerabilities for existing source code. This is combined by an architecture recovery approach [93] from the co-authors of the paper.

The developed approach is illustrated in Figure 4.14. The basic concept is that a static code analysis extracts from the build configuration the dependencies.

In our case, we use Snyk[10] as a static analysis. The analysis then compares the dependencies to their list of vulnerable components and returns for each build artefact a list of security vulnerabilities. Our new approach (black box in Figure 4.14) then uses these security vulnerabilities and extracts the CVE from it. Based on the extracted CVE, it queries the NVD via its *Rest-API* for the CVSS metric. It then creates a vulnerability model by using our vulnerability metamodel.

For mapping the created vulnerability to PCM elements, we used the architecture recovery approach from Kirschner [93] with his trace link model. The trace link creates a link between the build artefact and the recovered components. So far, this approach is limited to `BasicComponents`. Other artefacts, such as the `ResourceContainers` or `LinkingResources`, cannot be recovered.

After creating the vulnerability model and linking it to the recovered components, the security experts have to enrich the model to use our attack propagation analysis. They need to create the missing architectural model, such as `Assembly`, `Deployment` and `ResourceEnvironment`. In addition, they need to enrich the model with access control information and, if necessary, adapt the vulnerability model. The adaption can be necessary since the NVD uses a more general version of a CVSS, and it might vary for different configurations.

Using this approach can help to reduce the effort for identifying existing vulnerabilities and modelling these. However, it does not help with unknown vulnerabilities. However, in the future, the approach might be extended to support these. In Kirschner et al. [94], we discuss the accuracy of the approach regarding different case studies.

4.4. Considering Attacks in Software Architecture Models

This section covers the modelling concept for attacks in software architectures. It is part of our contribution C2 together with the vulnerability metamodel

[10] URL: `https://snyk.io/` (visited on 11/02/2021).

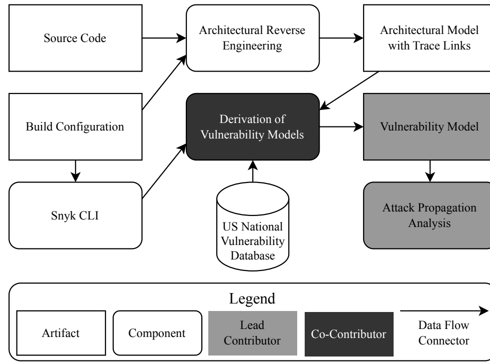


Figure 4.14.: Approach for automatic extraction of vulnerabilities based on [94]

(c.f. Section 4.3) and attacker metamodel (c.f. Section 4.5). The contribution C2 and C1 answer our research question **RQ2.1**. In addition, the contribution C2 is part of the foundation to answer research question **RQ2.2**. We published this metamodel part in Walter et al. [211].

The previous sections described the vulnerability metamodel and how it enables an architect to assign vulnerabilities to architectural elements. The vulnerabilities already have information about how to exploit them and what the impact of the exploitation is. However, the actual exploitation is not described or modelled. The exploitation is usually performed through an attack. In our modelling, these attacks represent also the activities a malicious user can perform to exploit a vulnerability. Therefore, we call these also the capabilities of an attacker.

In this section, we will first describe the requirements for our attack metamodel in Section 4.4.1. Afterwards, we describe in Section 4.4.2 our attack metamodel.

4.4.1. Requirements for Modelling Attacks

Like in the previous modelling sections, we identified different requirements, which guided the development of our metamodel. As previously done, we will first list the identified requirements and afterwards discuss the reasoning. Our identified requirements are the following.

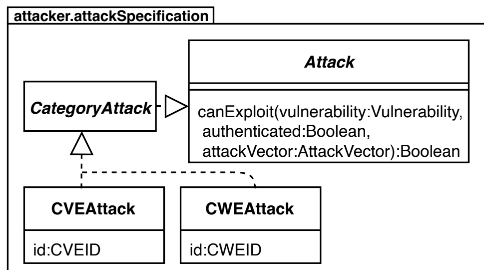


Figure 4.15.: Attack metamodel elements based on [211]

ATR1 Security experts should be able model attack which conform to the vulnerability metamodel.

ATR2 Security experts should be able to model concrete and non-concrete attacks.

The first requirement can be derived from our intended purpose of our attack modelling. The purpose of the attack part of our metamodel is to express attacks exploiting the modelled vulnerabilities. Therefore, it is necessary to be compatible to our vulnerability metamodel because security experts need to use the modelled vulnerabilities. Hence, **ATR1** states that the attack needs to be compatible with the described vulnerability metamodel elements. Otherwise, there is no connection between the modelled attacks and vulnerabilities.

The second requirement **ATR2** can be derived based on **ATR1** and the similar requirement **VR2** from the vulnerability metamodel. Because the vulnerability metamodel contains concrete and non-concrete vulnerabilities, the attacks need to handle concrete and non-concrete attacks. Another benefit of supporting non-concrete attacks is to express groups of attacks and therefore ease the modelling since not every single attack needs to be modelled.

4.4.2. Modelling Attacks

The capabilities of attackers are modelled as different attacks they can perform. The attacks are also represented in our developed metamodel. Figure 4.15 shows the involved metamodel elements.

The basic idea is to reuse the concepts from CVE and CWE again. These concepts are already used in the vulnerability part and widely adopted in the security domain for vulnerabilities. For simplicity reasons, we explicitly decided against using attack modelling concepts like the Common Attack Pattern Enumerations and Classifications (CAPEC) [38]. While these dedicated attacker description can be useful, they were not necessary for our analyses and would, in our eyes, only complicate the model creation. Therefore, we chose to use a more simplistic approach and reuse the CVE and CWE parts. This results in our two concrete attack elements which are the `CVEAttack` and the `CWEAttack`. Each element has an `id` attribute representing a reference to their respective id element. The `id` can be used later to match whether the attack could exploit a vulnerability. The `CVEAttack` can only exploit vulnerabilities with the same `CVEID`. This represents the possibility to model concrete attacks. The `CWEAttack` models the capability to exploit groups of attacks. A `CWEAttack` can exploit a vulnerability with the same `CWEID`. In addition, it can exploit all vulnerabilities with a child `CWEID`s from its `CWEID` and all vulnerabilities with their `CVEID` belonging either to its `CWEID` or any child `CWEID`. For instance, an attack with the `CWEID` “*CWE-312*” can exploit among others `CWEVulnerabilties` with “*CWE-312*”, “*CWE-313*”, “*CWE-314*”. In addition, it also can exploit among others the `CVEVulnerability` “*CVE-2021-28374*”. By using the attacks in this way, we also use fulfil **ATR2** and **ATR1** since the concrete `CVEVulnerabilities` and the non-concrete `CWEVulnerabilities` can be exploited by our attack metamodel elements.

Each attack inherits from `CategoryAttack` and `Attack`. The `CategoryAttack` encapsulates all attacks, which are based on a categorisation. Currently, these are the only attacks, the metamodel supports. The `Attack` is the abstract element for all attacks. It defines a boolean operation which can check whether a `Vulnerability` can be exploited by an `Attack`.

4.5. Considering Attackers in Software Architecture Models

This section explains our metamodel for attackers in software architectures. It is part of our contribution C2 together with the vulnerability metamodel (c.f. Section 4.3) and attack metamodel (c.f. Section 4.4). The contribution C2

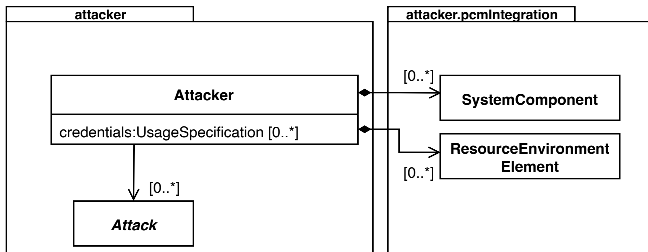


Figure 4.16.: Attacker propagation metamodel elements based on [211]

and C1 answer our research question **RQ2.1**. In addition, the contribution C2 is part of the foundation to answer research question **RQ2.2**.

So far our metamodel can express the vulnerabilities and the attacker capabilities required for exploitation. In combination with the access control metamodel from Section 4.2, a software architect can also model the access control policies. However, the metamodel does not cover so far dedicated malicious users. In our metamodel, we call these malicious users attackers. In our case, attackers have capabilities and knowledge. Capabilities are for us the attacks they can perform and knowledge is useful information they have about the system, such as credentials.

In this section, we will first describe a dedicated attacker for attack propagations in Section 4.5.1. Afterwards, we describe in Section 4.5.2 an attacker for filtering different attack paths.

4.5.1. Modelling Attackers for Attack Propagation

The first attacker type is for an attack propagation. The main concept is that an attacker has an initial starting point in the architecture. Based on this starting point the attacker can propagate to other elements by using their knowledge and capabilities. This behaviour is similar to insider attacks or system breaches, where the attacker propagates from one breach point through the system. We initially published this attacker in Walter et al. [211].

Figure 4.16 illustrates the metamodel elements for this attacker type. The central element is the `Attacker`. The capabilities of the attacker are explic-

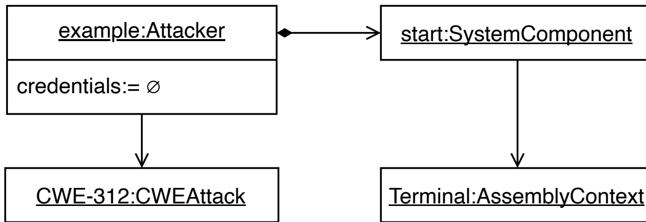


Figure 4.17.: Attacker propagation instance for the running example

itly and implicitly modelled. The explicit capabilities are the `Attacks`. An `Attacker` can have multiple or none `Attacks`. The implicit capabilities are the capability to use the knowledge of credentials and automatically compromise components on compromised hardware. For the first, the knowledge is modelled as a list of `UsageSpecifications` representing credentials.

The start point is represented by contained `SystemComponents` and `ResourceEnvironmentElements`. These are reference elements into the PCM (c.f. Section 4.3.4). The start point can cover multiple elements, for instance, in case of multiple breach points.

A possible attacker for our running example can look as in Figure 4.17. This model describes a scenario, where an attacker somehow got access to the `Terminal` and uses it as a starting point for an attack on the system. In this case, the `Attacker` has no initial knowledge about credentials, but can perform a `CWEAttack` for "*CWE-312*". This attack describes, that an attacker can use credentials stored in cleartext. Hence, we assume it is a reasonable capability an attacker might have. The start point for the attacker is set by the `SystemComponent` which references the `Terminal` component.

4.5.2. Modelling Attackers for Filtered Attack Paths

The first attacker type is useful if the security expert has already decided on a starting point and the skill set of the attacker, for instance, with insider attacks or system breaches. However, if the security experts are interested in whether, for a certain architecture, an attack path exists and the concrete capabilities of the attacker are unknown, then our second attacker type can be used. This attacker type is initially published in Walter et al. [212].

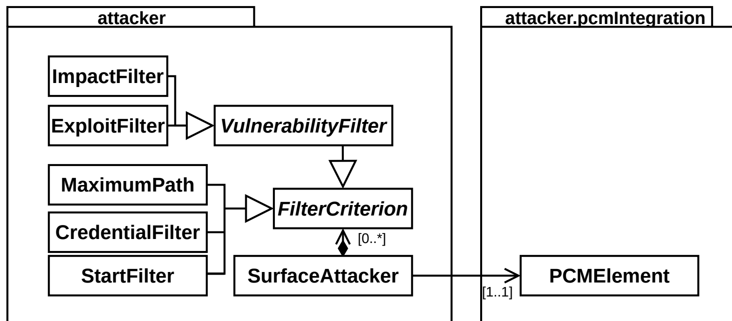


Figure 4.18.: Attacker with filtered attack paths metamodel elements

The second type is cornered around attack paths and filtering of these. Figure 4.18 illustrates the metamodel elements. In contrast to the previous attacker model, the knowledge and the capabilities are not explicitly modelled. The metamodel only defines restrictions for the space of capabilities and knowledge. These restrictions are called `FilterCriterion`. The abstract `FilterCriterion` is contained by the `SurfaceAttacker`, representing our second attacker type. A `SurfaceAttacker` can have multiple or none `FilterCriterions`. In addition, the `SurfaceAttacker` defines a target for the attack paths by referencing a `PCMElement`. While the model allows any of the referenced architectural elements, the analysis only supports `AssemblyContexts`. The start points for the attack paths are determined by the analysis unless a filter explicitly selects one.

The `FilterCriterions` can be used to select attack paths. This selection is necessary since multiple different attack paths can lead to the targeted element. We have five concrete elements for the criterion. They are separated between criteria regarding the used vulnerabilities in the attack path with the abstract `VulnerabilityFilter` and criteria regarding the path in general. We will first explain the general filters and then the ones related to the vulnerabilities.

- `MaximumPath`: This filter sets a maximum length for an attack path. Usually, attack paths in a large system can contain multiple elements and get very huge. Therefore, also the calculation for every attack path increases. However, in some cases, experts want faster results and therefore restrict the solution space to attack paths with a maximum

path length. In other cases, experts are interested only in short paths since they are only interested in a subsystem around the targeted element. Also, related approaches, such as Polatidis et al. [148, 147], use a similar filter criterion and define it as one of their requirements.

- `CredentialFilter`: This filter allows experts to limit the initial credentials used in an attack path. Attackers can use credentials or vulnerabilities to propagate through a system. Therefore, an attack path could consist of vulnerabilities or user credentials. However, the credentials are knowledge an attacker needs. Attackers need to get this knowledge somehow. It can be either that they already have the knowledge at the start or that they gain it during the attack. An example of the first case is initial knowledge through other non-modelled channels, such as insider knowledge. For the second case, an example is through exploiting vulnerabilities which can leak certain credentials. For instance, the vulnerability in our running example leaks the *admin* attribute. Our analysis assumes that an attacker can have any credential at the beginning. Hence, with the filter, we can restrict the initial knowledge. An alternative design decision is that we explicitly model the initial knowledge. However, in this attacker type, we decided to use always restrictions.
- `StartFilter`: The `StartFilter` restricts the start point for an attack path. During the second analysis, for each architectural element, an attack path to the targeted element is calculated. This calculation might be very time-consuming. Hence, experts might want to restrict the calculation similar to the `MaximumPath` to get faster results. This can be done with the `StartFilter`, where experts can select the start components. This can be useful, for instance, if an expert is only interested in attacks from certain parts of the system. Based on our running example, an example scenario is, that an expert is only interested in attacks from the `Terminal` to the targeted `ProductStorage` since the `Terminal` is externally accessible. Therefore, it is more exposed than the other elements. Also, other approaches use the concept of a dedicated starting point together with a dedicated endpoint [148].

The second type of filters is modelled around the properties of the used vulnerabilities. The common upper element is the `VulnerabilityFilter`. These filters are:

- `ImpactFilter`: This filter is build on the impact metrics from CVSS. We use these metrics also in our `Vulnerability` element. The idea is, that security experts can filter for attack paths using only, for instance, vulnerabilities with a high impact on confidentiality. Hence, in the element experts can select the minimum value for the confidentiality impact. Similar to the confidentiality impact, the filter also supports filtering based on the availability and integrity impact. For both, the expert can select the minimum impact. For instance, setting the availability impact as a minimum on `High` would return attack paths, which only contain vulnerabilities with a high availability impact. The ordering for each attribute is illustrated below. We start with the lowest value to the highest value:
  - *Confidentiality impact:* None < Low < High
  - *Integrity impact:* None < Low < High
  - *Availability impact:* None < Low < High
- `ExploitFilter`: This filters based on the exploitability metric from CVSS. The `Vulnerability` element also uses similar metrics.Here, the idea is that the security expert can set a maximum value, for instance, for the attack complexity and then the attack path would only contain vulnerabilities with this set value as a maximum. This concept is again used to reduce the solution space for attack paths. Therefore, it can fasten up the calculation and filter for only relevant attack paths. The metamodel supports the filtering for the attack vector, attack complexity, required privileges, and user interaction. An example based on our running example is, that security experts during a security assessment are only interested in attack paths with a `Low` attack complexity since they assume that attackers will not invest a lot of effort for the selected targeted element. The ordering for each attribute is illustrated below. We start with the lowest value to the highest value:
  - *Attack vector:* Network < AdjacentNetwork < Local
  - *Attack complexity:* Low < High
  - *Privileges:* None < Low < Special
  - *User interaction:* None < Required

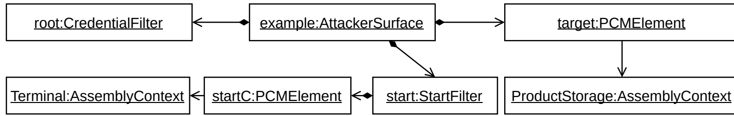


Figure 4.19.: Attacker instance with filters for the running example

In general, all these filters can help to reduce the solution space for identified attack paths and therefore reduce the calculation time. In addition, they can help to identify relevant attack paths. The number of architectural elements as a starting point in larger systems can be big. Therefore, the number of attack paths can be big. Security experts might not be able to identify the relevant ones. However, the filter option can help to identify the relevant ones.

An example scenario for our running example is illustrated in Figure 4.19. It describes a scenario where security experts are interested in attack paths from the `Terminal` to the `ProductStorage` without the initial knowledge about the `root` credentials. For this, we use a `AttackerSurface` with two filters. The first is the `CredentialFilter` for the `root` credential. The second is the `StartFilter`, which selects by using a `PCMElement` the `Terminal`. The target is set by a contained `PCMElement` to the `ProductStorage`.

5. Analysing Software Architectures for Potential Security Incidents

The introduced metamodel provides security experts with the means to model access control properties, vulnerabilities, attacks and attacker and annotate them to software architectural elements. The pure modelling of these properties does not help to identify potential security incidents. However, the modelling can be used as a documentation and act as a foundation for analysing the software architecture regarding potential security incidents. This analysis can happen manually by experts, which study the modelled properties and can decide based on their knowledge and experience what incidents can arise. Hence, the results of the manual analysis depend on the knowledge of the experts. Even more in larger and more complex system architectures, it gets harder for experts to get a complete understanding of the system. In these cases an automatic analysis can help to get more complete results.

In this work, we developed three different automated security analyses based on our introduced metamodels. The first analysis is described in Section 5.1. It uses our contribution C1. The analysis forms our contribution C3 and answers our research question **RQ1.2**. The contributions is a scenario-based usage analysis for access control policies. Based on a usage scenario, we analyse whether service calls are possible with the current policy model or whether violations are identified.

The second analysis is an attack propagation analysis. It builds up on our contributions C1 and C2. The attack propagation analysis is our contribution C4.1. The contribution answers the research question **RQ2.2** regarding the attack propagation. The analysis investigates the propagation from an attacker starting from an initial architectural element and using vulnerabilities and access control policies to compromise new architectural elements. It uses

a dedicated attacker model (c.f. Section 4.5.1) with knowledge and concrete capabilities. This analysis is described in Section 5.2.

In Section 5.3, we describe the third analysis. It calculates potential attack paths to a targeted architectural element. It uses our contributions C1 and C2. The third analysis is our contribution C4.2 and answers our research question **RQ2.2** regarding the attack paths. The used attacker model (c.f. Section 4.5.2) provides different filter criteria to restrict the design space of the attacker.

5.1. Scenario-Based Access Usage Analysis

This section introduces our scenarios-based access usage analysis, which is built up on our contribution C1. The analysis is our contribution C3 and answers our research question **RQ1.2**. This analysis is built up on our publication Walter et al. [214].

Understanding the impact of access control policies is complicated. The access in context-based access control policies like in our ABAC-based access control metamodel can depend on various different attributes, which change dynamically. For instance, in our running example the access to the scenarios depends on the dynamically changing state of the machine. Understanding the access decision and potential impact is getting in larger systems even more complicated, where service calls are often delegated between different components and involve different services or users. One approach to estimate the impact is by analysing the intended usage in different scenarios and identifying potential access violations. Similar approaches are performed by Seifermann et al. [168] for data flow analysis and Boltz et al. [32] for handling uncertainty in an access control data flow analysis. Verma et al. [204] call similar approaches experimental validation. In our work, we use a similar approach using different scenarios with system calls and access control policies. These scenarios represent the intended usage or misusage of the system by users. We developed a scenario-based access usage analysis, which investigates whether the intended usage or misusage represented by the scenarios is possible with the current access control policies. For realising this, we extended the scenarios with context information (c.f. Section 4.2.4) and our developed analysis investigates for each modelled service in a scenario whether it is possible or not with the current context of the scenario. The analysis determines the result based on the specified access control policies.

The result is a list of potential access violations and can help to identify potential security incidents.

5.1.1. Process for Analysing Scenario-based Access Control Policies

Our **RQ1.2** investigates how we can identify whether certain scenarios are possible with the given software architecture and access control policies or not. Because of the focus on scenarios and access control policies, we decided to develop a scenario-based access analysis.

The main concept of the scenario-based access analysis is to analyse whether different usage scenarios are possible with the modelled access control policies. In PCM, the usage is modelled in the usage model with different `UsageScenarios`. PCM defines, that the `UsageScenarios` are modelled by the *Domain Expert* [154, p. 24]. The Domain Expert has knowledge about the intended use cases and how users use or intend to use a system. In our extension, we will keep this role. The other roles defined by PCM, such as the *Component Developer* or *Software Architect*, are summarized in our process as the role *Software Architect*. As previously described, we assume that the *Software Architects* cover the modelling of the software architecture. The different roles are important for the general design of the software architecture. However, for the explanation of our process, they are not that important. As introduced in the motivation, we also define the new role *Security Expert* to the predefined PCM roles. The *Security Expert* has knowledge about the specification of access control policies and about requirements for confidentiality. Similar approaches, such as the data flow analysis by Seifermann [166] or the process for alignment of access control roles by Pilipchuk [143], also define the role of security experts, who models security-related properties in PCM. Similar to the related approaches, our role is responsible for the access control properties and the definition of attributes. In contrast to the other approaches, the role also helps in the definition of misusage scenarios.

Figure 5.1 illustrates our intended process for a new software project (greenfield development). The process syntax is based on an extended UML activity diagram syntax. We have the three described roles `Security Expert`, `Software Architect`, and `Domain Expert`, each indicated by its respective icon. For each activity (boxes with round corners), we assign the involved

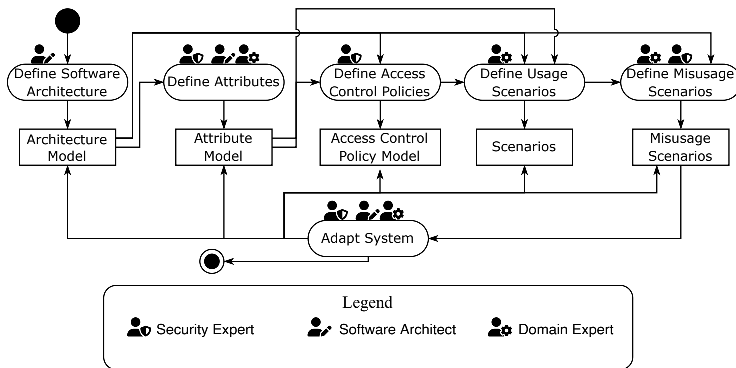


Figure 5.1.: Process for using the access usage in a new system.
Icon Source: Font Awesome by Dave Gandy — `http://fontawesome.io`

roles by putting their icon on it. The involved artefacts are the generated models (boxes with sharp corners).

The process starts by defining the initial software architecture through the `Software Architects`. They define the components, system, deployment and hardware resources. They can model these manually or if artefacts already exist, use existing reengineering approaches, such as SoMoX [25], Monschein et al. [119], or Kirschner [93].

Afterwards, all roles are involved in the creation of the attribute model. This is performed by all three roles since it requires knowledge from each role. The attribute model covers the different user roles. These depend on the role model from the `Security Expert`, but they need the knowledge from the `Domain Expert`, who knows which users will use the system and what properties they have. In addition, the attribute model will involve properties emitted from different components, such as the machine state in our running example. This system knowledge stems from the `Software Architects`. Hence, they are involved. The attribute model and the software architecture are then used by the `Security Expert` to create the access control policies.

The `Domain Experts` define based on the software architecture and the attributes the different usage scenarios. They use the software architecture to identify the service calls used by a user and the attribute model to describe the context during a scenario. The context can be a user's role or a machine state.

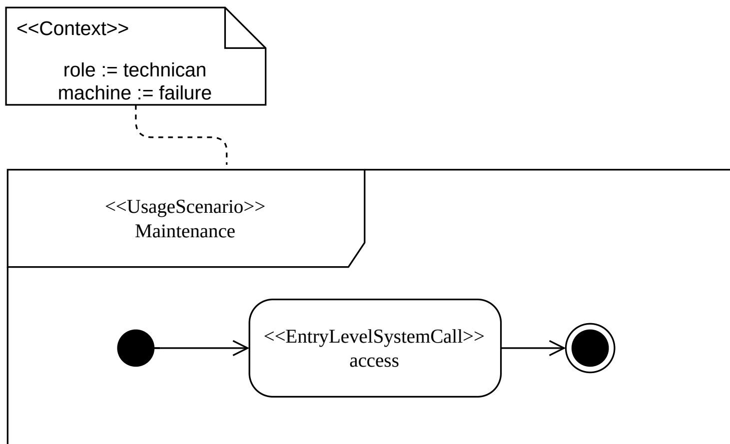


Figure 5.2.: UsageScenario for the running example with the technician accessing the log data of the machine during a failure state

Figure 5.2 illustrates a modelled usage scenario based on our running example. PCM follows here the syntax of UML activity diagrams. A `UsageScenario` contains a start and end point. Between them, different activities are performed. In our case, it is a call to the `access` service of the system. In PCM, these system calls are specified by the `EntryLevelSystemCall`. A usage scenario is not limited to one `EntryLevelSystemCall` like in our example but can contain multiple ones to different services of the system. The context for the scenario is set by the attributes for the whole scenario as illustrated by the comment (box connected with the dashed line) in Figure 5.2. In the figure, we use the comment box only as a simplified representation. In the approach, the context is a dedicated model element (c.f. Section 4.2.4). The modelled scenario in Figure 5.2 illustrates a context where the requestor has the role technician and the machine is in a failure state.

Besides the scenarios, which model the intended usage of a system, our access analysis also supports misusage scenarios. Similar to the concept of *misuse cases* [177] and *mal-activity diagrams* [176], they model usage scenarios, which should be prohibited. The idea is that the modelled policies are not only evaluated against the intended usage but also evaluated whether they can prohibit misusage. For instance, a misusage scenario in our running example can be that the service technician tries to access the machine without

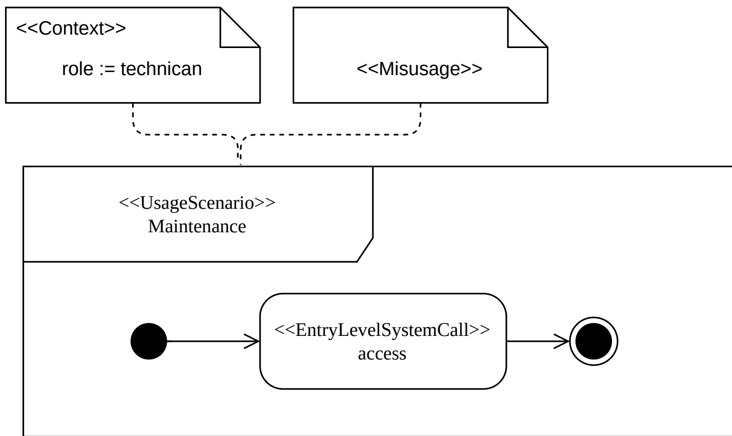


Figure 5.3.: MisusageScenario for the running example with the technician accessing the log data of the machine

a failure state. Figure 5.3 illustrates this scenario. In contrast to the previous scenario, the machine state is missing and the annotation `Misusage` is added. The annotation is represented by a UML comment (dashed lines) and marked with a stereotype declaration of «`Misusage`» in Figure 5.3. The usage of the comment box is only used as a simplified representation in the figure. The approach uses a dedicated model element (c.f. Section 4.2.4). The annotation marks the usage scenario as a misusage scenario. The usage and misusage scenarios are created by the `Domain Expert` together with the `Security Expert`. Both roles should be involved since the `Domain Expert` can provide insights about the usage and the `Security Expert` about identifying misusage.

The last step is the potential adaption of the system after the analysis is performed. The input for the last activity are all created models. However, we left this out for a better readability of the figure. All three roles investigate, whether the results of the analysis are sufficient. If the results are sufficient, the process ends. If not, they can adapt the models to get the desired results.

This process should not be considered a strict procedure. It should act more as a showcase and guideline on how our analysis is integrated into the component-based development process defined by PCM. For instance, some activities can also be done in parallel or each role can be further defined, like a security expert for policy design and misusage detection. It is also

possible to adapt the process in evolution processes or iterative development processes. A simple adaption could be that the definition is only an adaption or the remodelling of an existing approach.

5.1.2. Analysing Scenarios for Access Violations

After the specification of the software architecture, the access control policies, the attributes and the scenarios, the access usage analysis can analyse each scenario for access violations. The analysis can detect for each scenario and each contained system call access violations.

The conceptual idea for our analysis is to analyse the service calls regarding access violations. We determine the service calls based on the usage scenarios which describe the initial service calls a user performs. Afterwards, our analysis follows the initial service calls and identifies all following service calls. For each of the following service calls, the analysis determines the current context and checks whether the context is sufficient to access the called service. In the end, the analysis has a list of services and the access decision. Based on this list, the analysis can determine whether a usage scenario is possible or not. In the following, we describe the analysis in more detail and give insights about the implementation. We slightly simplify the explanation of the implemented access usage analysis to make it easier to follow. However, we will give insights about the simplification and the complete implementation is available in our dataset [208].

The analysis is described using the metamodel elements from Chapter 4. Using the metamodel elements for the implementation allows us to reuse the same algorithm for different system instances. Instances of these metamodel elements can be mapped to sets. These sets are then representing the elements. For example, all service instances can be grouped into a set. In this section, we only define a simplified set containing only the relevant sets for the scenario-based access usage analysis. In the later section, we extend these sets for the other analysis. We define the following sets:

- $US = \{us_1, \ldots, us_n\}$ set of all usage scenarios
- $S = \{s_s, \ldots, s_n\}$ set of all `Services`
- $A = \{a_1, \ldots, a_n\}$ set of all attributes
- $P = \{p_1, \ldots, p_n\}$ set of all policies

Based on these sets, we can define different boolean functions. Each function checks certain metamodel properties. In case the conditions are met, each function returns true otherwise false. The different parameters can be overloaded. Each function is defined first by the name, then by the parameters. Afterwards, a logical expression is following, describing the check conditions. In the following, we will first introduce the function and its definition and then will provide an optional description.

- $isContext(a, u) := a \in A \land u \in US \land a$ context of u
  Checks if a is a context attribute of the usage scenario u.
- $isContext(a, s_{origin}, s_{target}) := a \in A \land s_{origin} \in S \land s_{target} \in S \land a$ is context between s_{origin} and s_{target}
  Checks if a is a context attribute between the services s_{origin} and s_{target}
- $fullfill(x, y) := x \subseteq A \land y \in S \land x$ grant access for y
  Checks whether the set of attributes grants access to the service y. In our analysis, this check is performed by querying the PDP with the loaded policies. The query process is described in Section 4.2.5.

Besides the different boolean checks for the different conditions, we also need a function, which provides a set of the connected services.

$$connectedServices(s) : s \in S \rightarrow next \subseteq S := \begin{cases} next \subseteq S, & \text{service s calls } next \\ & \text{other services} \\ \emptyset, & \text{no other service is} \\ & \text{called by s} \end{cases}$$

The idea of the function is that it provides a set of services which are called by the passed service parameter. The analysis requires this because services can delegate calls to other services.

Based on the defined sets and functions, we illustrate the analysis for one usage scenario in Algorithm 1. In case multiple usage scenarios should be analysed, the algorithm can be repeatable called. The algorithm expects the attributes with the set `A`, a usage scenario with the parameter `scenario`, and the services with the set `S`. In the case of our analysis, the usage scenario needs to be annotated as discussed in Section 4.2.4. In the end, the algorithm returns the result for the scenario, and it contained service calls, which is a tuple of the access decision and the service.

Algorithm 1 Simplified Scenario-Based Access Usage Analysis

1: **procedure** ACCESSUSAGEANALYSIS($A, scenario \in US, S$)
2: $\quad result := \emptyset$
3: $\quad$ **for all** $s_{init} \in scenario$ **do**
4: $\quad\quad context := \{a \in A \mid isContext(a, s_{init})\}$
5: $\quad\quad$ **if** $context = \emptyset$ **then**
6: $\quad\quad\quad context := \{a \in A \mid isContext(a, scenario)\}$
7: $\quad\quad$ **end if**
8: $\quad\quad result = result \cup (fullfill(context, s), s)$
9: $\quad\quad nextServices = connectedServices(s_{init})$
10: $\quad\quad$ **for all** $s_{next} \in nextServices$ **do**
11: $\quad\quad\quad result = result \cup analyseService(s_{next}, s_{init}, context, A)$
12: $\quad\quad$ **end for**
13: $\quad$ **end for**
14: $\quad$ **return** $result$
15: **end procedure**

The first step in Algorithm 1 (l. 2f) is to initialise the result set. The algorithm initialises it as an empty set. Later on, the algorithm will fill it iteratively with tuples of service instances and access decisions. The next step (l. 3) is to iterate over all the services in the scenario. As described in Section 5.1.1, a scenario can contain multiple calls to services. In the case of PCM these service calls from a usage scenario are called `EntryLevelSystemCalls`. The next step is to identify the context for the service call. The context is the attributes assigned to the scenario or the service call in the usage scenario (c.f. Section 4.2.4). In the algorithm, we first derive in l. 4 the context set for the called service. If this set is empty, we select the context of the usage scenario. This checking for an alternative context enables us to override the general context of a usage scenario. However, the design decision of directly replacing the context if one is modelled comes with the drawback that an empty context for a service cannot override the context of a usage scenario. An alternative design decision, which use, for instance, a dedicated flag for overriding would solve this problem. We choose to use the simpler approach because it does not require additional modelling elements.

After the algorithm determined the context, it checks whether the current context has access to the selected service (l. 8). Very simplified, the function checks whether the current context is enough to access the service. Math-

ematical formulized, it checks whether $(context \cap policiy) = policy$ where $context \subseteq A$ describing the context of the call and $policiy \subseteq A$ describing the policy protecting the service. In the actual implementation, this decision is made by the PDP. There, we need to create a XACML request as described in Section 4.2.5 and then query the PDP. Because of the usage of XACML the access decision is in the implementation also much more complicated. In our description, we described only a basic feature. In reality, the PDP cannot perform a simple set comparison in all cases because XACML can have arbitrary comparison functions for the requested set. We refer for a more detailed explanation to our limitations (c.f. Section 7.4 — *Access Control Model*) and the XACML documentation [132]. After determining the access, we save the result together with the service in the `result` set.

Often a service calls other services. These services then again call other services. Hence, a service call in a usage scenario has something like a call stack of following service calls. This is similar to regular programming, where functions can call other functions, thereby, creating a call stack. Each of these service calls can have access control protection. Therefore, the analysis needs also to analyse the subsequent service calls for violations. We realise this analysis of the subsequent service calls by first identifying the directly called services. Afterwards, we iterate over the called services and call for each one the `analyseService`. Then, we add the return to the result set and return it in the end.

Algorithm 2 Simplified Analysis of Service Calls

1: **procedure** ANALYSESERVICE($s \in S, predecessor \in S, context \subseteq A, A$)
2: $\quad context_{new} := \{a \in A \mid isContext(a, s, predecessor)\}$
3: $\quad$ **if** $context_{new} \neq \emptyset$ **then**
4: $\quad\quad context := context_{new}$
5: $\quad$ **end if**
6: $\quad result = result \cup (fullfill(context, s), s)$
7: $\quad nextServices = connectedServices(s)$
8: $\quad$ **for all** $s_{next} \in nextServices$ **do**
9: $\quad\quad result = result \cup analyseService(s_{next}, s, context, A)$
10: $\quad$ **end for**
11: $\quad$ **return** $result$
12: **end procedure**

The `analyseService` is described in Algorithm 2. The parameters are the current service to analyse represented by `s`, the calling service represented by the parameter `predecessor`, the current context, which consists of attributes represented by the parameter `context`, and the set of all attributes represented by `A`. The first step in the function is to determine the context for the service call. The context can be either the same context as the predecessor or it is overridden by `AttributeProviders` (c.f. Section 4.2.4). For determining the context, we need to consider the current service and the predecessor service because, based on our modelling, the context is defined on the relation between the two services. In the case a new context is available for the service, we set the new context ($context_{new}$) as the context. The rest of the function is similar to the Algorithm 1 for the service calls from a usage scenario. We check whether the context grants access to the selected service and save the outcome in the `result` set. Then, we identify possible called services and recursively analyse these services.

In the end, Algorithm 1 and Algorithm 2 have recursively analysed all the service calls within a usage analysis and return the result set. The set contains the access decision for each service calls.

After obtaining the `result` for a usage scenario, we iterate over the `result` and check whether all contained service calls' results are possible. If this is the case, we mark the scenario as passed and continue with the next scenario. This process is repeated until no more scenarios exist. In the end, the analysis determined for each scenario, whether it is passed or not.

Remark: For simplicity reasons, we assumed during the explanation that the access decision *true* is the case when access is possible and *false* is the case when access is denied. The actual implementation by the PDP uses a more fine-grained result (c.f. Section 4.2.5). In the implementation, *Permit* represents the boolean *true* value from our description and all the other possible results are *false*.

5.1.3. Analysing Misusage Scenarios for Access Violations

The misusage scenarios are the scenarios containing the unwanted usage of the system. Due to our modelling decision to reuse the existing scenarios for the misusage, we can reuse most of the aspects from the scenario-based access usage analysis. The main modelling difference is that we annotate

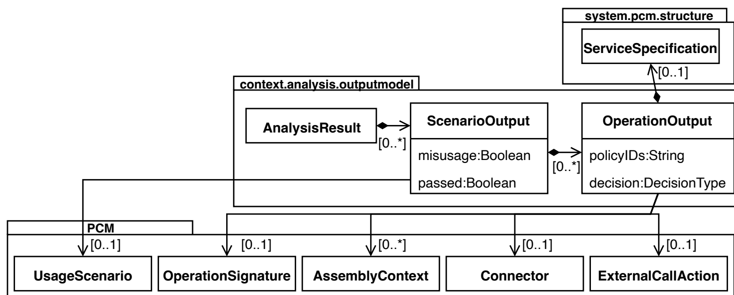


Figure 5.4.: Result metamodel for the access usage analysis

a usage scenario with `Missusage` (c.f. Section 4.2.4). The meaning is that the scenario should not be possible. We define a scenario as impossible if it contains at least one service call which is denied.

This comes with the drawback that we cannot explicitly check whether certain service calls are impossible. The analysis always determines the pass decision for the full scenario. However, this problem can be circumvented by using smaller scenarios where the number of called services is smaller. Despite this drawback, the benefit is that from the technical side, we can reuse the previously described scenario-based access analysis (c.f. Section 5.1.2). For each marked misusage scenario, the analysis performs Algorithm 1. Afterwards, the analysis also iterates over the `result`. In contrast to the non-misusage scenario, we mark the scenario only as passed if at least one access result is false (meaning `Deny` in the implementation). Otherwise, the scenario is marked as not passed. This way, it stores the same result as for the other scenarios, and we can reuse most parts of the analysis.

5.1.4. Result Model for the Access Usage Analysis

In this section, we describe the result model for the access usage analysis. The result model is the output of the analysis. It can be used in the adaption step of our process to decide whether the system is satisfying or not. We choose to define the result models also by a metamodel.

Figure 5.4 illustrates the metamodel for the analysis results. The root container element is the `AnalysisResult`. It contains for each usage and misusage

scenario a `ScenarioOutput`. As attributes, it has a reference to the `Usage-Scenario` in PCM it represents, a boolean flag indicating whether it is a misusage scenario or not, and a boolean flag indicating whether the scenario is passed or not. It is a misusage scenario if the `misusage` attribute is `true`. Otherwise, it is a usage scenario. The scenario is passed, when the `passed` attribute is `true`.

Each `ScenarioOutput` contains a list of `OperationOutputs`. These are the different access decision results for the called services in a scenario. It is based on the `result` from Algorithm 1. Each `OperationOutput` has a string list with policy IDs and the decision of the PDP is stored in `decision`. The datatype is the same as the result model for the PDP decision (c.f. Section 4.2.5 and Figure 4.10). Besides these elements, it contains an optional `ServiceSpecification` to represent the called system services. The access decision for the `EntryLevelSystemCall` is encapsulated by the references to various PCM elements. These are an `OperationSignature`, a list of `AssemblyContext`, a `Connector` and an `ExternalCallAction`. These are necessary to identify the actual called service on the instance level and not on the type level as in the `EntryLevelSystemCall`.

Figure 5.5 illustrates the result model for our running example. It shows an excerpt of our tooling with the available editors. The upper half shows an EMF tree editor for a result model. We see the different `ScenarioOutputs` and whether they are passed (indicated by `true`). The first `ScenarioOutput` is the scenario where the service technician accesses the machine during a failure state. This further information can be found in the lower half of Figure 5.5 in the *Property View* (indicated with the text `Properties`). Each `ScenarioOutput` contains then the `OperationOutput` with the access decision. In the first case, it is `Permit`. The software architect, security expert or domain expert can then use this result model. They can click on the different elements and see more detailed information from the metamodel. This can help them to decide whether the architecture, together with the access control policies, is suitable for the intended usage or not. If they decide it is not useful, the information provided by the result model could be a first starting point for changes.

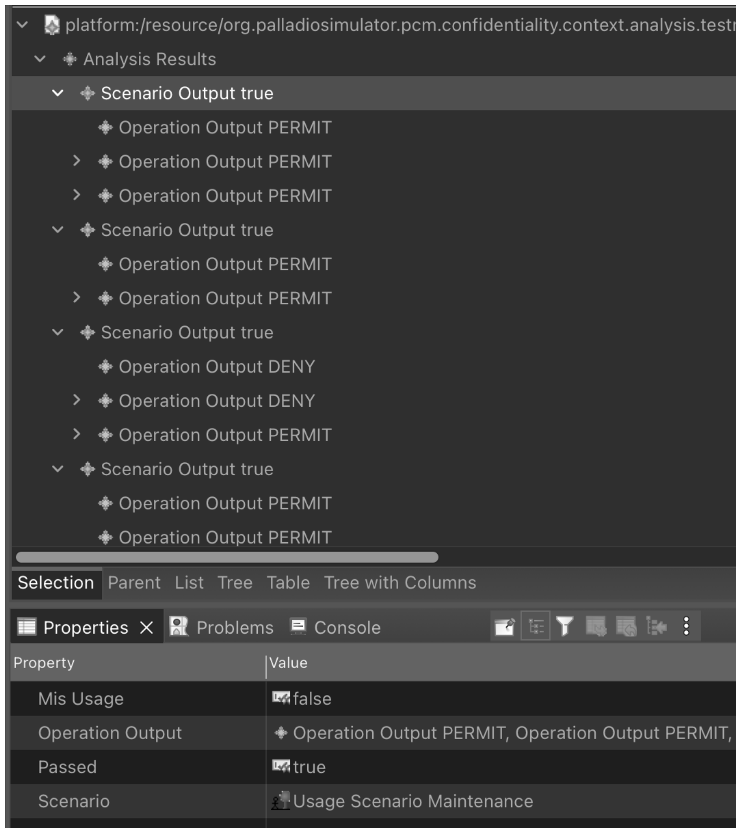


Figure 5.5.: Analysis result model for the running example

5.2. Attack Propagation Analysis

In this section, we introduce our attack propagation analysis. It builds up on our contributions C1 and C2. The analysis is our contribution C4.1 and covers the first part of the research question **RQ2.2**. The other part is answered by C4.2. We originally published the analysis in Walter et al. [211] and extended or used it in other publications [209, 213, 94].

Our second type of analysis is settled around an attack propagation based on the software architecture. The access analysis (C3) works well for scenarios where only access control properties are relevant. It can consider malicious users with the misusage scenarios, but the analysis is limited to the intended control flow of services. More complex scenarios, which do not work along the control flow of the system, cannot be modelled and analysed. For instance, in our running example, an attacker tries to propagate from the `Terminal` directly to the `StorageServer`. In doing so, the attacker would not follow the different service calls in the system. In this dependence on the intended control flow, the attacker behaviour in C3 is similar to an honest-but-curious or semi-honest attacker behaviour [139] in protocol design, meaning that attackers cannot deviate from the original protocol. Here, the intended control flow can be seen as a protocol. To support this deviation from the control flow, we need another type of analysis, which can propagate along other structural properties.

Another aspect not considered in the access analysis is the impact of vulnerabilities and their interplay with access control policies. A software system can contain multiple vulnerabilities which attackers may exploit. However, often it is unclear whether attackers can actually exploit them. For instance, a vulnerability might not be reachable for an attack or the attacker needs certain privileges to exploit them. In addition, it is often unclear what the impact on the overall system is. It might be that an attacker can only compromise non-essential parts of the system or very confidential data is affected. For instance, in our running example, the `ProductStorage` is considered very critical compared to the other components. In other words, compromising the `ProductStorage` is worse than the compromisation of the other components. Therefore, compromising it has a more significant impact than compromising another component. Another aspect of the impact is whether a vulnerability enables the easier propagation of an attacker, either by exploiting another vulnerability or by providing credentials for architectural elements.

We will first explain in Section 5.2.1 the process for using our attack propagation analysis. Afterwards, we explain the analysis process in Section 5.2.2. The difference between the two sections is that the first describes the analysis from a user view, and the second describes the internal steps the analysis needs to perform to analyse a system. We describe how we identify the affected data in Section 5.2.3. This concept of the extraction is then used in our attack propagation. The concept and the algorithm of the attack propagation is introduced in Section 5.2.4. The analysis creates a dedicated result model,

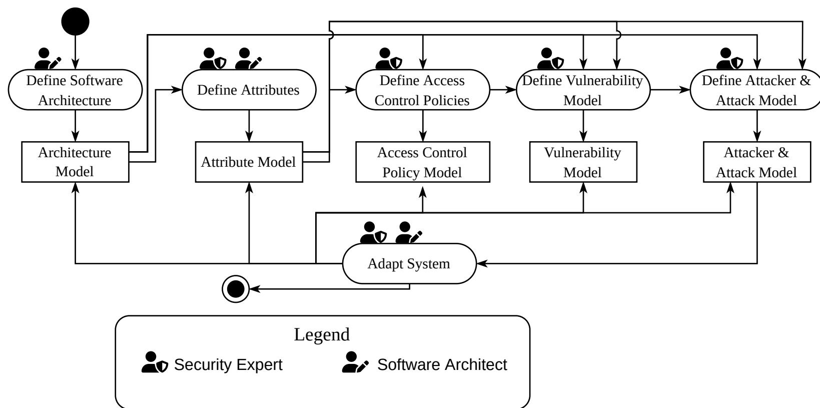


Figure 5.6.: Process for using the attack propagation in a new system.
Icon Source: Font Awesome by Dave Gandy — `http://fontawesome.io`

which is based on a result metamodel we developed. We introduce this result metamodel in Section 5.2.5.

5.2.1. Process for Analysing Attack Propagations with the Software Architecture

Similar to the access usage analysis, we describe an exemplary process for using the attack propagation analysis. The process is illustrated in Figure 5.6. The attack propagation does not use the usage models, which are created by the `Domain Expert`. Hence, in this process `Domain Experts` are not necessary. Nevertheless, for other PCM analyses, they are required. We kept the role of `Software Architect` the same but modified the `Security Expert`'s responsibilities. In this analysis, the role is also responsible for vulnerabilities, in addition to the definition of access control properties. This further differentiates the role `Security Expert` from similar roles in related PCM-based security analyses.

The first three steps in the process are identical to the process from the access usage analysis. This is beneficial since the different models can be reused between the different analyses. This can reduce the modelling effort

since only one `Architectural Model`, `Attribute Model` and `Access Control Policy Model` for both analyses needs to be modelled.

The fourth step is to create the vulnerability model based on the proposed metamodel (c.f. Section 4.3.3) by the `Security Expert`. This process can be manual or semi-automated. For the manual part, `Security Experts` can use existing databases or use their experience to model them. For the semi-automated part, they can use our developed approach in Kirschner et al. [94] (c.f. Section 4.3.5).

The next step is the creation of the attacker model by the `Security Expert`. Here, the security expert can define the attack and the propagation attacker based on our metamodels elements for attacks (c.f. Section 4.4.2) and the propagation attacker (c.f. Section 4.5).

The last step is the system adaptation based on the analysis results. The analysis evaluates the architecture and identifies affected architectural elements from an attack propagation. Based on these results, the `Security Expert` and `Software Architect` can change the created models to mitigate certain attacks. By mitigating attacks, they can reduce the potential set of affected architectural elements. Mitigation tactics are, for instance, stricter access control policies, exchanging vulnerable architectural elements with non-vulnerable ones or introducing mitigation approaches for certain vulnerabilities, such as using a Firewall to limit the access to the elements. They can also decide that the results are sufficient for the system. For instance, such cases are if either no attack propagation is possible or the system can only be compromised by very rare and costly attack types.

The resulting attack paths are specific to the given architecture. However, the approach and some models can be reused for other component-based architectures. We elaborate on the potential reuse in the evaluation (c.f. Section 7.2.4). Similar to the process of the access usage, this process should act as a guideline and does not need to be followed strictly. For instance, some steps could be done in parallel or performed by multiple persons with the same role. Examples of this additional separation could be using dedicated security experts for access control policies and vulnerabilities.

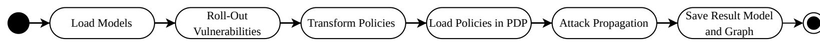


Figure 5.7.: Attack propagation analysis steps

5.2.2. Attack Propagation Process

The actual analysis process is then straightforward. We illustrate it in Figure 5.7. The process is split into different steps, which process the data. This happens similarly to the workflow definition of other PCM analyses. The first step is to load the different models. Here, we can reuse already existing PCM workflow actions.

Afterwards, we roll out the vulnerabilities to the instance level. Our metamodel allows the annotation of `BasicCompoents` and `AssemblyContexts`. The first is an annotation on the type level. However, the analysis requires the vulnerabilities on the instance level. We needed the type-level integration for the automatic derivation of vulnerabilities (c.f. Section 4.3.5). The handling of the type levels can be realized by extending the vulnerability look-up in the analysis itself. This would require an additional vulnerability check that additionally checks the `BasicComponent` for vulnerabilities. In our case, we decided against it, because it requires code changes within the analysis and negatively affects the runtime for the analysis itself. Therefore, we provided a new roll-out phase in our analysis workflow. This step annotates all the `AssemblyContexts` with a vulnerable `BasicComponent` with the vulnerabilities of the `BasicComponent`. This provides the analysis with the vulnerable `BasicComponents` without modifying the internal analysis behaviour. A drawback is that during the analysis, it is unclear whether vulnerability originates from a `BasicComponent` or the `AssemblyContext`. However, this information can be manually deduced by checking whether the corresponding `BasicComponent` of an `AssemblyContext` is vulnerable with the same vulnerability.

After the roll-out of the vulnerabilities from the `BasicComponents` to the `AssemblyContexts`, we transform the access control model into XACML policies (c.f. Section 4.2.5) and load these into our PDP. Afterwards, we execute the attack propagation analysis. The last step is to save the result model. Here, the user can choose between an EMF-based output and an additional graph representation.

5.2.3. Data Extraction

Besides the identification of affected architectural elements, also the identification of potentially stolen information is relevant for estimating the impact of attacks. This is especially relevant for considering the confidentiality because if attackers get access to data, it can be a confidentiality issue. For instance, if the stolen data contains sensitive information such as credit card data or other personal data, the confidentiality can be violated. Examples of attacks with stolen credit card data are, for instance, the attack on *HomeDepot*[1] or *Target*[2]. However, also other data can be considered confidential, such as in our running example, the data managed by the `ProductStorage`. In our case, information consists of different aspects, such as information about the software architecture or the managed data in an architecture. In this section, we focus on the last aspect, the managed data.

The managed data is usually described in PCM as parameters and return values of services. Other approaches, such as Werle et al. [217] or Seifermann [166], extended the data capabilities of PCM with more data-specific operations. However, as their foundation, they still use the parameters and return values of operations. Hence, we choose in this work also to use this datamodel. The benefit is that we use an established modelling technique in PCM. In addition, it eases the reuse of existing architectural models. The drawback is that we are limited to services for the data extraction. Hence, data is only indirectly modelled for components or hardware devices since components provide services and components are deployed on hardware resources. Even more, network resources have no data at all. In contrast, in reality, network elements like switches or routers can have some minimal data and even more they could see exchanged data over the network. However, this might not be too critical since it can be assumed that most of the critical data is stored or processed in services. In addition, some of the additional data, which can have an influence on the confidentiality or be beneficial for the attackers, are separately modelled. For instance, credentials are modelled

[1] J. Finkle et al. *Home Depot breach bigger than Target at 56 million cards.* Sept. 18, 2014. URL: `https://www.reuters.com/article/us-home-depot-dataprotection-idUSKBN0HD2J420140918` (visited on 06/06/2023).

[2] B. Krebs. *Inside Target Corp., Days After 2013 Breach.* Sept. 21, 2015. URL: `https://krebsonsecurity.com/2015/09/inside-target-corp-days-after-2013-breach/` (visited on 09/01/2021).

by the `AttributeProviders`. These can also be assigned to network elements, hardware resources or components.

Our analysis assumes that an attacker has complete control of the data if they either compromise an architectural element or exploit a vulnerability with a `Medium` or `High` confidentiality impact. If this is the case for a service, we extract the data by using the parameters with which it is called and the return values of external calls for service calls. For an affected component, we calculate it for every service provided by the component and for an affected resource container, we calculate it for each component deployed on it.

The extracted data can be used to estimate the criticality of an attack propagation. For instance in Walter et al. [209], we combined the extracted data with a data flow analysis to estimate the criticality of the affected data. There, our attack propagation provided the affected data, and the data flow analysis provided an estimation of criticality for each data object.

5.2.4. Analysing Attack Propagations

The core of the attack propagation analysis is the propagation algorithm. It illustrates how attackers could propagate through a system by performing attacks on vulnerabilities and exploiting access control properties. The propagation concept is built up on the KAMP approach [36]. KAMP has been already successfully used together with the PCM [155, 156]. As previously described, the core idea in KAMP is to propagate change requests by defined propagation rules over the structural elements of a software architecture (c.f. Section 2.1.3). The approach focuses on change propagations during maintenance tasks and not like our work on attack propagation. Nevertheless, we assume that the basic propagation concept is similar to our approach because the propagation in KAMP follows along the structural connection of the software architecture. It creates iteratively the transitive closure of the affected elements. This iterative process is repeated till no new elements are affected by the propagation. The same behaviour can be observed by attacks, which follow along the connected architectural elements. Instances for such connections are the deployment relationship between hardware devices and components or the network connections between hardware devices. As previously described, the attacker also propagates the attacks until no new element is affected, meaning no new element is compromised. However, our

attack propagations approach differs in the following aspect from the existing KAMP approach:

I) Different propagation types in the propagation rules.

II) Usage of dynamic propagation conditions for propagation rules.

The first difference is based on the maintenance focus in KAMP. The existing propagation rules do not consider the possible propagation on component instances. They are written for the type level of components, such as `BasicComponents`. In contrast, in our work, we need to consider the propagation on the instance level because, in our case, an attacker compromises the instances of the components. Therefore, we also annotate our vulnerabilities mainly on the `AssemblyContexts` which represent instances of components. Therefore, we needed to define new propagation types and propagation rules. Nevertheless, there can be cases where the propagation of attackers on types is useful, such as in supply chain attacks where attackers compromise the type itself. However, these attacks and propagations are out of the scope for our thesis.

The second difference is the usage of dynamic conditions in the propagation rules. In our propagation, we have dynamic properties, such as the knowledge of the attackers or the roles of compromised elements which change during a propagation. However, these dynamic changing properties can affect the result and need to be considered in the propagation rules. For instance, attackers can gain new knowledge about an attribute. This can enable the attackers to compromise a previously uncompromisable element. Therefore, our propagation rules need to consider these dynamic attributes in the propagation.

Because of these two differences, we cannot reuse the existing propagation rules and need to adapt the propagation framework. Our adaptation consists of the newly added propagation types for attacks and new propagation rules. In addition, we slightly adjusted the workflow to include our metamodels. Overall, we defined 19 new propagation rules considering the different propagation mechanisms of attackers. We explain these rules later in more detail during the explanation of the attack propagation (c.f. Algorithm 3 and Algorithm 4).

In the following, we describe a slightly simplified version of the implemented attack propagation, which is easier to follow. The main difference to the actual implementation is the gained attributes. We will describe the difference in the

part about gained aspects in more detail. Similar like the previous scenario-based access usage analysis, the attack propagation is described using the metamodel elements from Chapter 4. As previously mentioned using the metamodel elements for the algorithm allows us to reuse the same attack propagation algorithm for different system instances. As in Section 5.1.2, we map instances of these metamodel elements to sets. These sets are then representing the elements. For instance, all `AssemblyContexts` elements are a set. In this section, we extend the previously in Section 5.1.2 defined sets. We add only the set necessary for our attack propagation. We add the following sets:

- $AC = \{ac_1, \ldots, ac_n\}$ set of all `AssemblyContexts`
- $RC = \{rc_1, \ldots, rc_n\}$ set of all `ResourceContainers`
- $LR = \{lr_1, \ldots, lr_n\}$ set of all `LinkingResources`
- $AE = AC \cup RC \cup LR$ Architectural elements for the analysis
- $PAF = AC \cup RC \cup LR \cup S$ Potential affected elements
- $V = \{v_1, \ldots, v_n\}$ set of all vulnerabilities
- $AT = \{at_1, \ldots., at_n\}$ set of all attacks
- $ATV = \{Network, Adjacent, Local\}$ set of attack vectors
- $AUTH = \{true, false\}$ set of authorization
- $ROLE = \{r_1, \ldots, r_n\}$ set of roles
- $NON_GLOBAL = \{x \in AC \mid NonGlobalCommunication\}$ set of all components marked with `NonGlobalCommunication`

Like in Section 5.1.2, we define different boolean functions based on these sets. We also reuse the introduced functions from Section 5.1.2. Each function checks certain metamodel properties. In case the conditions are met, each function returns true otherwise false. The different parameters can be overloaded. Each function is defined first by the name, then by the parameters. Afterwards, a logical expression is following, describing the check conditions. In the following, we will first introduce the function and its definition and then will provide an optional description.

- $isAttackable(x, y) := x \in AT \wedge y \in V \wedge x$ exploits y
  Checks whether x can be used to exploit y. It can be exploited when the CWE or CVE from the attack match the CWE or CVE from the vulnerability. The exact description for matching can be found in Section 4.4.2.

- $isVulnerable(x, y) := x \in PAF \wedge y \in V \wedge x$ is vulnerable to y
  Checks if the architectural element x is vulnerable to the vulnerability y. This is the case, if the element is annotated with the vulnerability and the `takeOver` is set to true.

- $fullfill(x, y) := x \subseteq A \wedge y \in PAF \wedge x$ grant access for y
  Checks whether the set of attributes grants access to the architectural element y. Here, we widen the definition to architectural elements in contrast to the definition given in Section 5.1.2. In our analysis, this check is performed by querying the PDP with the loaded policies. The query process is described in Section 4.2.5.

- $grants(x, y) := x \in AE \wedge y \in A \wedge x$ gives y
  Checks whether the architectural element stores the attribute y and can provide these to an attacker. This is using our concept of `Attribute-Providers` (c.f. Section 4.2.4).

- $deployedOn(x, y) := x \in AC \wedge y \in RC \wedge x$ deployed on y
  Checks whether the component x is deployed on the `ResourceCon-tainer` y. This uses the allocation model from PCM.

- $assembly2Global(x, y) := x \in AC \wedge y \in AC \mid \exists r \in RC \mid deployedOn(x, r) \wedge \exists rc \in RC \mid (\exists l \in LR \mid resourceLinking(r, l) \wedge resourceLinking(rc, l) \wedge rc \neq r \wedge deployedOn(y, rc) \wedge x \notin NON_GLOBAL \wedge y \notin NON_GLOBAL$
  Checks whether the component x is connected to the component y by checking whether both components are in the same network. This can be done by using the `LinkingResources` and checking whether both components are not marked with `NonGlobalCommunication` (c.f. Section 4.3.4).

- $assembly2Assembly(x, y) := x, y \in AC \wedge (x$ connected by `Assembly-Connector` with $y \vee assembly2Global(x, y))$
  Checks whether two components are connected. Components are connected, if there is a `AssemblyConnector` between them or they are globally connected over a `LinkingResource`.

- $assembly2Linking(x, y) := x \in AC \land$
  $y \in LR \land \ \exists r \in RC \mid deployedOn(x, r) \land resourceLinking(r, y)$
  Checks whether the component x is connected to the network element y. This is performed by identifying the hardware resource, which deploys the component and then checking whether the hardware resource is connected with the network element.

- $assembly2Resource(x, y) := x \in AC \land y \in RC \land \neg deployedOn(x, y) \land \exists c \in AC \mid c \neq x \land assembly2Assembly(x, c) \land deployedOn(c, y)$
  Checks whether the component x is connected with the hardware resource y. This is done by identifying connected components and then using the deployment relation in the allocation model to get their hardware resources. This explicitly does not consider the hardware resource on which x is deployed.

- $resourceLinking(x, y) := x \in RC \land y \in LR \land x$ connected by link with y
  Checks whether the hardware resource x is in the same network as the network node y.

- $provides(x, y) := x \in S \land y \in AC \land y$ provides x
  Checks whether the component y provides the service x.

- $connectedService(x, y) := x \in AC \land y \in S \land \neg provides(y, x) \land \exists c \in AC \mid c \neq x \land provides(y, c) \land assembly2Assembly(x, c)$
  Checks whether the service y is connected to the component x. This can be done by identifying, the connected components.

- $resourceConnectedResource(x, y) := x, y \in RC \land x \neq y \land$
  $\exists l \in LR \mid resourceLinking(x, l) \land resourceLinking(y, l)$
  Checks whether two hardware resources are connected. Two hardware resources are connected if both are connected to the same `LinkingResource`.

- $isVulnerable(x, y) := x \in ATV \land y \in V \land x$ can exploit y
  Checks if the `AttackVector` x can be used to exploit the `Vulnerability` y. The description, when this is fulfilled can be found in Section 4.3.3 regarding the vulnerability's attributes.

- $isVulnerable(x, y) := x \in AUTH \land y \in V \land x$ can exploit y
  Checks if the vulnerability y needs an authenticated attacker to exploit it or not. Authenticated is indicated by the value `True`.

- $isRole(x, y) := x \in R \land y \in AE \land y$ provides role x
  Checks whether the architectural element y provides the `Role` x in the software architecture (see also Section 4.3.4).
- $isVulnerable(x, y) := x \subseteq R \land y \in V \land x$ compromises y
  Checks whether the `Vulnerability` y needs a compromised `Role` x, so that the vulnerability can be exploited (c.f. Section 4.3.3).

Besides the different boolean checks for the different conditions, we also need a function to map two architectural elements to an `AttackVector`. We define it as the function *atv*. It takes two architectural elements (*AE*) and returns the `AttackVector` (*ATV*). The `AttackVector` is `Local` if the two elements are in a deployment relationship, e.g., x is deployed on y or vice versa. If both elements are in the same network, meaning they are connected with the same `LinkingResource` (c.f. *resourceConnectedresource* and Section 4.3.3), it returns `AdjacentNetwork`. Otherwise, the value is `Network`.

$$atv(x, y) : AE \times AE \rightarrow ATV := \begin{cases} Local, & \text{deployment.} \\ AdjacentNetwork, & \text{in the same network.} \\ Network, & \text{otherwise.} \end{cases}$$

The attack propagation algorithm is then described in Algorithm 3 and Algorithm 4. For space reasons, we needed to split the algorithm into two parts. The main concept for the attack propagation is that it iteratively calculates the transitive closure of the directly attacked and indirectly attacked architectural elements. In addition, an attacker can gain new knowledge regarding the credentials (here attributes) in each iteration. The algorithm's termination condition is that no changes occur in the current iteration. This behaviour is based on KAMP. Hence, the structure was partly given by the foundational approach.

Besides the software architecture annotated with vulnerabilities and access control policies the input for Algorithm 3 is:

- $START$: A set of initial starting points in the software architecture. These can be any element from *AE*. It can be multiple elements. In a real attack, this would symbolize the initial break point.
- KL: A set of `Attributes` representing the initial knowledge of the attacker. In our approach, we use these to represent credentials or other attributes used as authorisation. Attackers can gain this knowledge

through *Social Engineering* attacks like *Phishing* or other attacks, which we do not consider in the analysis itself. However, the results of these attacks can be considered as knowledge. In addition, the initial knowledge can be empty. During the analysis, attackers can gain additional knowledge.

- *CAP*: A set of `Attacks` describing the capabilities of an attacker.

The output of the algorithm is a set of affected architectural elements (PAF).

The first step in the algorithm is to assign the $START$ set to N, which is the return value. Afterwards, it determines the roles of the initial compromised elements. Then the algorithm calculates iteratively the transitive closure of the compromised elements. It terminates if no new elements are affected in an iteration step, i.e., the attacker did not compromise a new element or gained no new knowledge. We determined the termination criteria by investigating which affected/compromised element can lead to potentially new propagations. In other words, compromised elements can open new attack paths for an attacker. The first criterion is affected architectural elements. An attacker can use these to reach new architectural elements which were previously not reachable. For instance, in our running example, the `ProductStorage` is not directly reachable from every component. However, an attacker can reach it by first compromising the `StorageServer`. The second criterion is the knowledge of an attacker. Adding new knowledge can also open new attack possibilities for an attacker. For instance, a non-vulnerable component can only be compromised by credentials, despite that it is reachable. However, by getting the knowledge about the credentials, an attacker could compromise it. Hence, we added this as a termination criterion. In theory, the role also would need to be termination criteria since if it changes, there could be new vulnerabilities an attacker could exploit. However, in our case, the role assignment is coupled to an architectural element, and an attacker can gain a role only by compromising an architectural element. Hence, when there is a new compromised role, there is also always a new compromised architectural element. Therefore, both compromisations happen in the same iteration step, and we can consider only the architectural element compromisation since the role compromisation is also covered.

After checking the termination criteria, the algorithm calculates the propagation steps, starting from line 5 in Algorithm 3. We calculate the propagation from each architectural element type to the other types and for each propagation type (vulnerability or credential). The separation between architectural

Algorithm 3 Simplified attack propagation algorithm 1/3 (continued in Algorithm 4)

Input: $START \subseteq AE, KL \subseteq A, CAP \subseteq AT$
Output $N \subseteq PAF$

1: **procedure** ATTACKPROPAGATION
2: $N = START$
3: $ROLE = \{r \in R \mid \exists n \in N \mid isRole(r,n)\}$
4: **while** $START \neq \emptyset \vee KL_{new} \neq \emptyset$ **do**
5: $ACACV = \{ac \in AC \mid \exists n \in N \mid n \in AC \wedge ac \notin N \wedge assembly2Assembly(n,ac) \wedge \exists v \in V \mid isVulnerable(atv(n,ac),v) \wedge isVulnerable(fullfill(KL,ac),v) \wedge isVulnerable(ac,v) \wedge isVulnerable(ROLE,v) \wedge \exists a \in CAP \mid isAttackable(a,v)\}$
6: $ACACC = \{ac \in AC \mid \exists n \in N \mid n \in AC \wedge ac \notin N \wedge assembly2Assembly(n,ac) \wedge fullfill(KL,ac)\}$
7: $ACLRCV = \{rc \in RC \mid \exists n \in N \mid n \in AC \wedge rc \notin N \wedge deployedOn(n,rc) \wedge \exists v \in V \mid isVulnerable(atv(n,rc),v) \wedge isVulnerable(fullfill(KL,rc),v) \wedge isVulnerable(rc,v) \wedge isVulnerable(ROLE,v) \wedge \exists a \in CAP \mid isAttackable(a,v)\}$
8: $ACLRCC = \{rc \in RC \mid \exists n \in N \mid n \in AC \wedge rc \notin N \wedge deployedOn(n,rc) \wedge fullfill(KL,rc)\}$
9: $ACRCV = \{rc \in RC \mid \exists n \in N \mid n \in AC \wedge rc \notin N \wedge assembly2Resource(n,rc) \wedge \exists v \in V \mid isVulnerable(atv(n,rc),v) \wedge isVulnerable(fullfill(KL,rc),v) \wedge isVulnerable(rc,v) \wedge isVulnerable(ROLE,v) \wedge \exists a \in CAP \mid isAttackable(a,v)\}$
10: $ACRCC = \{rc \in RC \mid \exists n \in N \mid n \in AC \wedge rc \notin N \wedge assembly2Resource(n,rc) \wedge fullfill(KL,rc)\}$
11: $ACLRV = \{lr \in LR \mid \exists n \in N \mid n \in AC \wedge lr \notin N \wedge assembly2Linking(n,lr) \wedge \exists v \in V \mid isVulnerable(atv(n,lr),v) \wedge isVulnerable(fullfill(KL,lr),v) \wedge isVulnerable(lr,v) \wedge isVulnerable(ROLE,v) \wedge \exists a \in CAP \mid isAttackable(a,v)\}$
12: $ACLRC = \{lr \in LR \mid \exists n \in N \mid n \in AC \wedge lr \notin N \wedge assembly2Linking(n,lr) \wedge fullfill(KL,lr)\}$
13: $ACSV = \{s \in S \mid \exists n \in N \mid n \in AC \wedge s \notin N \wedge connectedService(n,s) \wedge \exists v \in V \mid isVulnerable(atv(n,s),v) \wedge isVulnerable(fullfill(KL,s),v) \wedge isVulnerable(s,v) \wedge isVulnerable(ROLE,v) \wedge \exists a \in CAP \mid isAttackable(a,v)\}$

Algorithm 4 Simplified attack propagation algorithm 2/3 (continued from Algorithm 3)

14: $ACSC = \{s \in S \mid \exists n \in N \mid n \in AC \wedge s \notin N \wedge connectedService(n,s) \wedge fullfill(KL,s)\}$
15: $LRACV = \{ac \in AC \mid \exists n \in N \mid n \in LR \wedge ac \notin N \wedge assembly2Linking(ac,n) \wedge \exists v \in V \mid isVulnerable(atv(n,ac),v) \wedge isVulnerable(fullfill(KL,ac),v) \wedge isVulnerable(ac,v) \wedge isVulnerable(ROLE,v) \wedge \exists a \in CAP \mid isAttackable(a,v)\}$
16: $LRACC = \{ac \in AC \mid \exists n \in N \mid n \in LR \wedge ac \notin N \wedge assembly2Linking(ac,n) \wedge fullfill(KL,ac)\}$
17: $LRRCV = \{rc \in RC \mid \exists n \in N \mid n \in LR \wedge rc \notin N \wedge resourceLinking(rc,n) \wedge \exists v \in V \mid isVulnerable(atv(n,rc),v) \wedge isVulnerable(fullfill(KL,rc),v) \wedge isVulnerable(rc,v) \wedge \exists a \in CAP \mid isAttackable(a,v)\}$
18: $LRRCC = \{rc \in RC \mid \exists n \in N \mid n \in LR \wedge rc \notin N \wedge resourceLinking(rc,n) \wedge fullfill(KL,rc)\}$
19: $RCACL = \{ac \in AC \mid \exists n \in N \mid n \in RC \wedge ac \notin N \wedge deployed(ac,n)\}$
20: $RCACV = \{ac \in AC \mid \exists n \in N \mid n \in RC \wedge ac \notin N \wedge assembly2Resource(ac,n) \wedge \exists v \in V \mid isVulnerable(atv(n,ac),v) \wedge isVulnerable(fullfill(KL,ac),v) \wedge isVulnerable(ac,v) \wedge \exists a \in CAP \mid isAttackable(a,v)\}$
21: $RCACC = \{ac \in AC \mid \exists n \in N \mid n \in RC \wedge ac \notin N \wedge assembly2Resource(ac,n) \wedge fullfill(KL,ac)\}$
22: $RCLRV = \{lr \in LR \mid \exists n \in N \mid n \in RC \wedge ac \notin N \wedge resourceLinking(n,lr) \wedge \exists v \in V \mid isVulnerable(atv(n,lr),v) \wedge isVulnerable(fullfill(KL,lr),v) \wedge isVulnerable(lr,v) \wedge \exists a \in CAP \mid isAttackable(a,v)\}$
23: $RCLRC = \{lr \in LR \mid \exists n \in N \mid n \in RC \wedge ac \notin N \wedge resourceLinking(n,lr) \wedge fullfill(KL,lr)\}$
24: $ASSEMBLY = ACACV \cup ACACC \cup LRACV \cup LRACC \cup RCACL \cup RCACV \cup RCACC$
25: $SER = \{s \in S \mid \exists n \in ASSEMBLY \mid s \notin N \wedge provides(s,n)\}$
26: $START = (ASSEMBLY \cup ACLRCV \cup ACLRCC \cup ACRCV \cup ACRCC \cup ACLRV \cup ACLRC \cup ACSV \cup ACSC \cup LRRCV \cup LRRCC \cup RCLRV \cup RCLRC) \setminus N$

Algorithm 5 Simplified attack propagation algorithm 3/3 (continued from Algorithm 4)

27: $N = N \cup START \cup SER$
28: $KL_{new} = \{x \in A \mid \exists n \in START \mid grants(x, n)\} \setminus KL$
29: $KL = KL \cup KL_{new}$
30: $ROLE = \{r \in R \mid \exists n \in START \mid isRole(r, n)\} \cup ROLE$
31: **end while**
32: **end procedure**

element types and propagation rules is beneficial because it enables a separation of the different propagation concerns. Another possibility could be the combination of propagation rules. However, then a propagation rule would not be responsible for only one propagation step, which can violate the coding principle of *Single Responsibility* [107, p. 95ff]. The propagation calculation is very similar for each. In all cases, we first identify the connected elements. Afterwards, we determine whether the attack can compromise the connected elements and, therefore, propagate to the connected elements. Here, we need to differentiate between two types. We decided to separate the propagation types because the handling of the different propagation types is different for each type. Hence, the separated rules are easier to define and should also be easier to maintain. The first type is the propagation by vulnerabilities. In this propagation, the algorithm checks whether the architectural element is vulnerable, the attack vector matches to the required attack vector from the vulnerability, the required roles are already compromised, the attacker has the correct authorisation level, and the attacker has a matching attack capability. If these are all fulfilled, then the attacker can propagate to the connected element. Hence, it is compromised. The second propagation type is the propagation by using credentials, i.e., exploiting the access control policies with gained attributes. Here, the algorithm verifies whether the attacker has the suitable attributes for accessing the connected element. The different propagation sets also indicate by their name the type and the involved elements. The last character indicates whether it is a vulnerability propagation by using V or by using C if it uses credentials. The involved types are shown by using the abbreviations symbols for the involved set. The first abbreviation illustrates the source type from which the propagation happens, and the second is the target type. For instance, for the first set

ACACV (l. 5 in Algorithm 3) it is `AssemblyContext` (AC) to `AssemblyContext` (AC) by using a `Vulnerability` (V).

In the following, the different propagation sets for each architectural element type and propagation type are calculated. Based on the description, we give them an order. Nevertheless, the order is changeable. By changing the order, obviously, the propagation within a loop iteration might be different. For instance, if the propagation of `AssemblyContexts` is handled after the `LinkingResources`. However, the results should stay the same because the changed compromised elements are only considered in the next step. Even if the changed compromised elements are directly considered, this is no problem because the missed propagation steps are redone during the next loop iteration.

The first propagation set *ACACV* (l. 5) represents the propagation between `AssemblyContexts` by exploiting a vulnerability. It first identifies all connected components and then whether they are vulnerable to the current properties of the attacker. The second set *ACACC* (l. 6) calculates the same, but uses the credentials. The third (*ACLRCV*, l. 7) and fourth (*ACLRCC*, l. 8) are the propagation from an `AssemblyContext` to the `ResourceContainer`, which deploys the `AssemblyContext` by using first vulnerabilities and secondly credentials. *ACRCV* (l. 9) and *ACRCC* (l. 10) are the propagation from `AssemblyContexts` to connected `ResourceContainers` using vulnerabilities and credentials. These sets do not cover the deployment relationship. We separated them since they are slightly different cases for the implementation. *ACLRV* (l. 11) and *ACLRC* (l. 12) are the propagation from `AssemblyContexts` to `LinkingResources`. The next two sets are *ACSV* (l. 13) and *ACSC* (l. 14) representing the propagation to connected services.

The algorithm is then continued in Algorithm 4 with the sets for the propagation from `LinkingResources`. The first two are *LRACV* (l. 15) and *LRACC* (l. 16) propagating to `AssemblyContexts`. The next two sets are *LRRCV* (l. 17) and *LRRCC* (l. 18) representing the propagation from `LinkingResources` to connected `ResourceContainers`. The next group of sets cover the propagation from hardware resources (`ResourceContainer`). The first set *RCACL* (l. 19) is the propagation from a `ResourceContainer` to a local `AssemblyContext`. Local means in our case, that the `AssemblyContext` is deployed on the `ResourceContainer`. In this case, we assume that the deployed `AssemblyContexts` are always compromised. The next propagation is the propagation to remote `AssemblyContexts`. These are `AssemblyContexts` not deployed on

the `ResourceContainers`. This propagation is done by *RCACV* (l. 20) and *RCACC* (l. 21). *RCLRV* (l. 22) and *RCLRC* (l. 23) are the propagations from `ResourceContainers` to `LinkingResources`. The set *ASSEMBLY* (l. 24) is the union of all propagation sets, which result in `AssemblyContexts`. The algorithm uses this set to calculate the propagation set for the services. These are the services from all affected `AssemblyContexts`.

Afterwards, the overall propagation sets for this iteration step are calculated. The next step is to determine the new $START$ set (l. 26). This set should contain, in the end, only newly affected elements so that we can use it as one of the termination conditions. If it also contains old elements, the algorithm does not terminate. Then we build the union of the new compromised elements and the old elements and assign it to the return value. In the next step, we update the attacker's knowledge and store the new knowledge in KL_{new} (l. 28). This set also contains only newly compromised attributes, which we archive by removing already-known attributes from the set. The algorithm updates the knowledge by identifying for the compromised architectural elements, which grant attributes. For simplicity reasons, we left out the second part of the knowledge update. In our implementation, we also store the traces for each compromisation, i.e., the source, the used attack and the used vulnerability or attributes. Based on this trace, we can also exploit in the implementation the `gainedAttribute` attribute from a `Vulnerability`. Hence, the result is more detailed. The last step is updating the compromised roles in the system. Here, the algorithm determines for each newly compromised element whether it acts as a role or not. Determining these sets is repeated till the termination criteria are fulfilled. In that case, the set N is returned, which contains the affected/compromised elements.

Based on the returned affected elements, we then can extract the affected data as described in Section 5.2.3.

We are now applying this concept to our running example. As a starting point, we select the `Terminal` and give the attacker the capability to perform attacks based on CWE-312 [114]. As knowledge, we select no initial knowledge about credentials. In Figure 5.8, we exemplary illustrate the propagation process as a graph. The nodes with only a label represent the compromised architectural elements. Nodes starting with *Role:* indicate attributes. The nodes with a component name and a method name indicate a server. For instance, `Terminal: access` is the access service from the `Terminal`. The other nodes containing a datatype are the extracted dataobjects, i.e., the

parameter and return values from the services. The edges contain the reason for the compromisation. This is either the used vulnerability, attribute or using one of our assumptions (indicated by implicit). If we now perform our attack propagation, the first affected element is the `Terminal` as the start point. Afterwards, the analysis checks each connected element for vulnerabilities or if we can use credentials. In the example, the only vulnerable connected element is the `TerminalServer`, and the attacker has so far no credentials they can use. The vulnerability of the `TerminalServer` is as previously described CVE-2021-28374 [129] and can leak the `admin` credentials. Also, our attacker has the capability CWE-312 [114], which is the correct capability to exploit the vulnerability. In addition, the analysis checks whether the attack vector is compatible. In this case, it is since the required one is `Network` (based on the CVSS) and the attacker has also `Network`. The next step is to check whether authorization is necessary or not. Based on the CVSS description, no authorization is necessary. Hence our attacker can fulfil this requirement. The last step is to verify whether a specific system role is necessary to exploit the vulnerability. In our case, we modelled it without that requirement since also the description does not mention it. In summary, the attacker fulfils all the necessary properties and can exploit the vulnerability. Since the vulnerability also leaks the `Admin` attribute, the attacker gains the knowledge of the `Admin` attribute.

In the next step, the analysis then checks all connected elements to the `TerminalServer` and `Terminal`. In this case, there are no new vulnerable elements available. However, with the knowledge gained from the previous step, the attack can get access to the `StorageServer` and `MachineController`. Other elements are not affected, and also no new knowledge is gained in this step.

In the next step, then the attack can propagate to the `AssemblyContexts` deployed on the `StorageServer` and `MachineController` resulting in the compromisation of the `ProductStorage`, `ProductionDataStorage`, `Machine`. Additionally, all the services provided by these components are compromised. Otherwise, no new elements are compromised or knowledge is gained.

In the next and last step, the analysis checks whether it can compromise any new elements or gain new knowledge. However, there are no new compromisations or knowledge, so the propagation terminates. Afterwards, the analysis also extracts the compromised data.

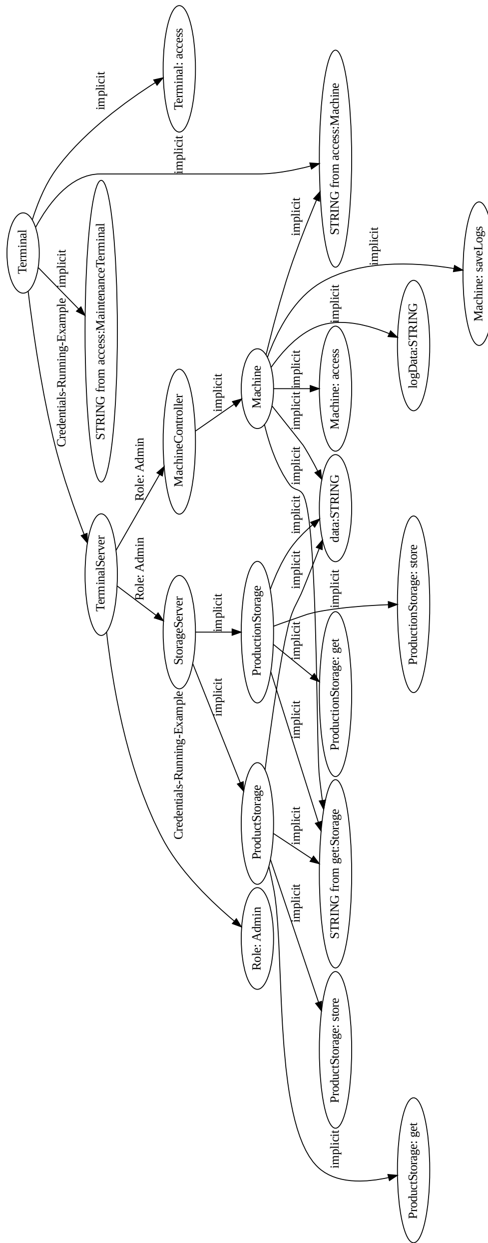


Figure 5.8.: Resulting attack propagation graph for the running example with the starting point `Terminal`

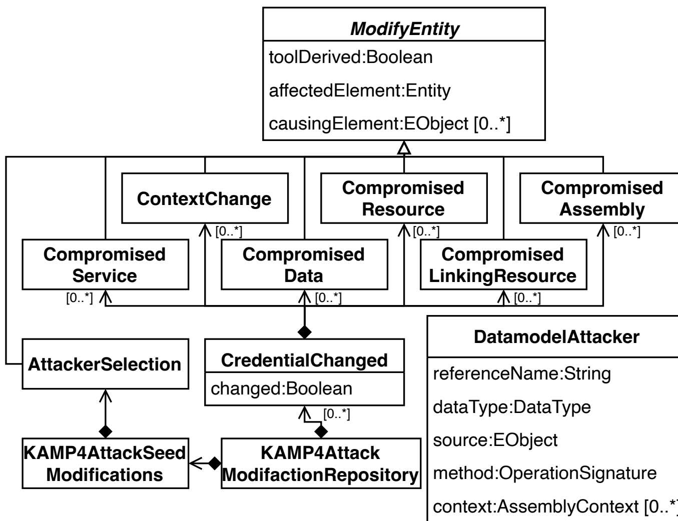


Figure 5.9.: Result and seed metamodel elements for the attack propagation analysis

5.2.5. Result Model for Attack Propagation

Based on the attack propagation analysis, software architects can have two result models. The first and mandatory result is a EMF-based model, which contains similar to a list of all affected elements. The second optional result is a graph like Figure 5.8.

The result metamodel is based on the *Modification Mark* metamodel from KAMP [36]. An excerpt of our metamodel is illustrated in Figure 5.9. The root element is the `KAMP4AttackModificationRepository`. It contains the start configuration (called seed in KAMP) and the result of a propagation. The seed is modelled with the `KAMP4AttackSeedModifications`. It contains a `AttackerSelection`, which references a `Attacker`. By selecting this as a seed, we configure the input for the attack propagation analysis since the `Attacker` contains the start point, the capabilities and the knowledge of an attacker (c.f. Section 4.5.1).

The result of the analysis is modelled with a `CredentialChanged`. It contains the child classes of the abstract element `ModifyEntity`, which represents a changed element in the original KAMP approach and a compromised element

in our approach. It has a boolean flag `toolDerived` indicating whether the element is derived from the analysis or not. For instance, the starting points are, in our case, not `toolDerived`. The `affectedElement` is the element which is compromised. In the figure, we choose to represent it as `Entity`. In the actual metamodel, this is bound to the concrete subclasses by EMF *Generics*. The last attribute is the `causingElement`. This is a list of `EObject` and contains the reason for the compromisation. This allows an architect to trace the compromisation back to a concrete reason. It contains the original component and the reason for the compromisation, i.e. the vulnerability or the used credentials.

The subclasses contain the reference to the architectural model elements. Each represents some elements an attacker compromised or the attacker itself. There are the six subclasses for compromised elements:

- `ContextChange` references a `UsageSpecification`
- `CompromisedResource` references a `ResourceContainer`
- `CompromisedAssembly` references a `AssemblyContext`
- `CompromisedService` references a `ServiceSpecification`
- `CompromisedData` references a `DatamodelAttacker`
- `CompromisedLinkingResource` references a `LinkingResource`

The `DatamodelAttacker` encapsulates the extracted data for the result metamodel. It contains the `referenceName`, which is the parameter name, and the `dataType`, which uses the defined PCM datatypes. These types are `BOOL`, `BYTE`, `CHAR`, `DOUBLE`, `INT`, `LONG`, `STRING`, `CollectionDataType` for collections/arrays, and `CompositeDataType` for custom data elements consisting of the other elements. The `source` element shows from where the element was compromised and `method` describes the service to which the parameter or return value belongs. The `context` attribute describes the involved components.

Figure 5.10 shows an instance of the result model in our developed tooling. The result model is for our running example. The tree editor (upper half of the figure) contains under the `CredentialChange` the list of compromised elements. The excerpt contains `CompromisedResource` elements, `CompromisedAssembly` elements and a `ContextChange`. The selected element is the `CompromisedResource` for the `StorageServer`. A software architect or security expert can see the details in the *Properties* views (lower half of the

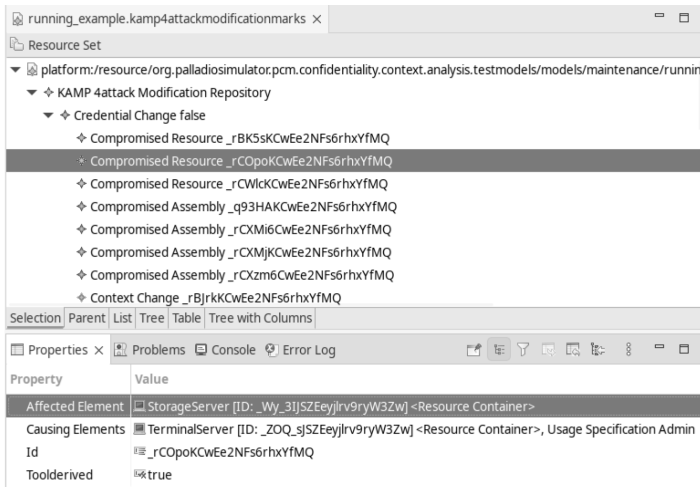


Figure 5.10.: Attack propagation result example for the running example in *Eclipse*

figure). First, it shows the affected element, in this case, the `StorageServer`. It shows the reason (causing elements). In this case, the attack originates from the `TerminalServer` and uses the `Admin` credential. In addition, a software architect or security expert can see the `ID` of the element and that the analysis created the element (`Toolderived`).

The second optionally result model is the graph presentation as shown in Figure 5.8. While the first model can show detailed information regarding the compromised elements, it is quite hard to understand the connection between compromised elements. The second model makes this easier with its graph representation. However, the graph is only a graphical representation without connections to the models. We create it by transforming each compromised element into two nodes connected with an edge. The first node is the source, the second node is the affected element, and the edge is named after the reason for the compromisation. Afterwards, we export these as a *dot* file, and *Graphviz*[3] connects the different edges and prints it. The graphical representation then helps to understand the connections between the attacks and compromised elements.

[3] *Graphviz*. URL: `https://graphviz.org/` (visited on 02/07/2023).

Software architects or security experts can then use the results to secure the system. For instance, in our running example, they can mitigate the attack by updating the `Terminal` to a non-vulnerable version. However, sometimes they cannot update the system, for instance, because of incompatibilities in the new version or if the `Terminal` is a legacy system which is already out of support. In these cases, a solution could be to change the access control policies for the other devices, so they require different attributes than the `Terminal` provides. Another possibility could be if they want to protect only the `ProductStorage` to deploy it to another more secured network and server. Overall, these are only some of the possible solutions. While our approach does not provide these solutions, it can help software architects or security experts to come to these solutions by showing the potential propagation of attackers through the system.

5.3. Targeted Attack Graph Analysis

In this section, we describe the attack graph analysis and attack path identification. The analysis uses our contributions C1 and C2. It is our contribution C4.2 and answers together with C4.1 our research question **RQ2.2**. We originally published the analysis in Walter et al. [212].

Our third type of analysis is an attack graph analysis. It creates an attack graph and can identify potential attack paths. In the previous attack analysis, we focussed on the propagation of attackers with concrete capabilities and knowledge. The result of the previous analysis is a concrete attack propagation graph. However, in some cases the concrete capabilities of attackers are unknown, or it is not relevant which elements an attacker can compromise, but rather whether there is an attack path to a concrete element. For instance, in our running example, a security expert might not be interested that the `Machine` is attacked, but more interested in whether the highly confidential `ProductStorage` can be attacked. In these cases, security experts or software architects can use our third analysis.

We will first explain in Section 5.3.1 the process of using this analysis and then in Section 5.3.2 the technical process. Afterwards, we explain in Section 5.3.3 the attack graph creation. This graph is then used to identify attack paths, as described in Section 5.3.4. In the end, we describe the result model of the analysis.

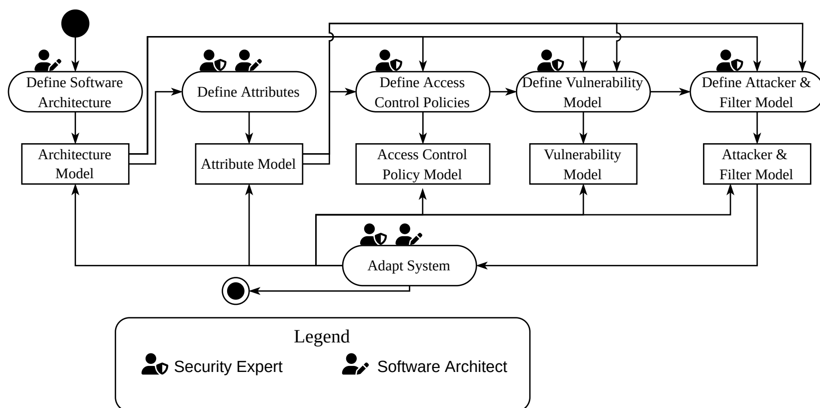


Figure 5.11.: Process for analysing attack graphs

5.3.1. Process for Analysing Attack Graphs based on the Software Architecture

Similar to the access usage analysis and attack propagation analysis, we describe an exemplary process for using the attack graph analysis. The process is illustrated in Figure 5.11. The process has the same roles as the attack propagation process (c.f. Section 5.2.1) and nearly the same steps.

The process also starts with the definition of the software architecture by software architects. Afterwards, the security experts define, based on the input from the software architects, the used attributes. The software architect is necessary because they know which attributes are available in the system due to the used components. These attributes are then used by the security experts in the specification of access control policies. The security experts also define the vulnerability model. So far, the steps are identical to the attack propagation process. Here, even the models can be reused. However, the next step differs from the attack propagation. In the previous process, the security experts need to define the concrete attacks and assign them to an attacker. In this case, the security experts do not define the concrete capabilities but define filters and assign them to attackers.

In the last step, the involved roles can again adapt the system to prevent potential security incidents. `Security Expert` and `Software Architect` can

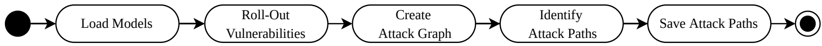


Figure 5.12.: Targeted Attack Graph analysis process steps

change the created models to mitigate certain attacks and break attack paths. Like in the attack propagation, mitigation tactics could be stricter access control policies, exchanging vulnerable architectural elements with non-vulnerable ones or introducing mitigation approaches for certain vulnerabilities. Besides introducing mitigation steps, they can also decide that the results are sufficient since, for instance, the involved attacks are very complicated or rare.

5.3.2. Attack Graph Analysis Process

The analysis process for the targeted attack graph is very similar to the process described for the attack propagation (c.f. Section 5.2.2). The first two steps (loading models, roll-out vulnerabilities) are identical and we reuse the same implementation. The next two parts are new. The *Create Attack Graph* step creates, based on the modelled software architecture, access control policies, vulnerabilities and filter criteria, an attack graph. The attack graph contains information on how attackers can compromise architectural elements. In the next step, the analysis uses the attack graph to identify attack paths leading to the targeted element. After the identification of the attack paths, the last step is to save the attack paths as the results of the analysis. For this part, we reuse existing concepts from PCM.

5.3.3. Creating an Attack Graph

The first part of our analysis is to create an attack graph based on the software architecture. The attack graph describes how attackers can compromise architectural elements. It is the foundation to determine the potential attack paths of attackers.

The graph creation is illustrated with Algorithm 6. In this part, we reuse the sets and functions defined in Section 5.2. The algorithm takes the software architecture, the vulnerabilities, the access control properties and the filter criteria and creates a multidimensional network, which is sometimes also

described as a labelled multigraph [41, p. 69]. This can be described as $G = (AE, E, D)$ where AE are the vertices, E the edges and D is the dimension [41, p. 69]. In our case, we define:

- $D = V \cup P \cup \{implicit\}$
- $E \subseteq V \times V \times D$ with $(v_i, v_j, d_k) \in E \wedge i \neq j \wedge v_i, v_j \in AE \wedge d_k \in D$

In other words, this means that our graph consists of architectural elements as vertices. The edges are a triplet. The first two values are the vertices between the edge. The last one describes the type of edge. The graph does not allow a self-loop. In our case, attackers can only take over the next architectural element in three cases:

1. in case of a vulnerability
2. in case of a policy granting access
3. in case of a deployment relationship (described as implicit)

In order for us to describe our attack graph creation, we need to define additionally the set $F = \{f_1, \ldots, f_n\}$, which is the set to describe the filter criteria. Besides the filter criteria set, we need these additional functions:

- $protectedBy(x, y) := x \in P \wedge y \in AE \wedge y$ protected by x
  Checks whether the architectural element is protected by a policy. In our, case these are sets of attributes.
- $notFilter(x, v) := x \subseteq F \wedge v \in V \wedge \nexists f \in x \mid f$ filters v
  Checks, that the `Vulnerability` v is not filtered by the set of filters.

After defining the required sets and functions, we describe the Algorithm 6 in more detail. The input sets of the algorithm are AE, V, P, F, and D. The output is the graph $G = (N, E, D)$ where $N = AE$. In other words, it takes the architectural elements, vulnerabilities, access control policies, filters, and dimensions and calculates an attack graph.

The first step of the algorithm is to initialise N and E. E is at the beginning empty, and N is a set of all architectural elements. The next goal is to iteratively create the edges and fill E with them. For this, we iterate over every architectural element and then identify for each element its neighbours (l. 5). The neighbours are all architectural elements, which are connected somehow to architectural elements. We consider them connected if one of the functions checking for a connection return true.

Algorithm 6 Simplified Attack Graph Creation

Input: AE, V, P, F, D
Output $G = (N, E, D)$

1: **procedure** ATTACKGRAPHCREATION
2: $N = AE$
3: $E = \varnothing$
4: **for all** $e \in N$ **do**
5: $neighbours = \{n \in N \mid n \neq e \wedge (deployed(n,e) \vee deployed(e,n) \vee assembly2Assembly(n,e) \vee assembly2Global(n,e) \vee assembly2Resource(e,n) \vee assembly2Resource(n,e) \vee assembly2Linking(e,n) \vee assembly2Linking(n,e) \vee resourceLinking(n,e) \vee resourceLinking(e,n) \vee resourceConnectedResource(n,e)\}$
6: **for all** $k \in neighbours$ **do**
7: **if** $deployed(e,k)$ **then**
8: $E = E \cup \{(e,k,implicit)\}$
9: **continue**
10: **end if**
11: $a = \{a \in P \mid fullfill(a,k)\}$
12: **if** $a \neq \varnothing$ **then**
13: **for all** $att \in a$ **do**
14: $E = E \cup (e,k,att)$
15: **end for**
16: **end if**
17: $V' = \{v \in V \mid isVulnerable(k,v) \wedge isVulnerable(atv(e,k),v) \wedge notFilter(F,v)\}$
18: **if** $v \neq \varnothing$ **then**
19: **for all** $v \in V'$ **do**
20: $E = E \cup \{(e,k,v)\}$
21: **end for**
22: **end if**
23: **end for**
24: **end for**
25: **end procedure**

Afterwards, the algorithm checks whether an attacker can exploit this connection. The first check is for the deployment relationship. If the neighbour k is deployed on the current architectural element, the algorithm adds an implicit edge. For instance, in our running example, there is an edge with `implicit` between the `MachineController` and `Machine`. If it is a deployment relationship, the algorithm continues with the next neighbour. Otherwise, the algorithm checks whether an access control policy protects the neighbour. If the neighbour is protected, the algorithm creates an edge for each policy. The same is repeated for the vulnerabilities. There, the algorithm first determines whether the vulnerability is applicable. It also considers whether the attack vector of the vulnerability is suitable for the exploitation. Besides checking the exploitability, the algorithm also checks whether the vulnerability is filtered. This takes advantage of the `VulnerabilityFilter` from Section 4.5.2. For example, an `ExploitFilter` with a selected high complexity would produce an attack graph that includes only vulnerabilities with low attack complexity. This is useful in scenarios where the software architects only want to consider attacks with a low level of complexity, for instance, to avoid unskilled attackers. A side benefit of the filtering in the graph creation is that the created graph gets smaller. A smaller graph means a smaller problem size for the latter path finding algorithms. Hence, it speeds up the process of finding attack paths. However, a possible drawback is that the created attack graph is not general but specific to the used `VulnerabilityFilters`. This could reduce the usability of the attack graph and might increase the overall performance in certain scenarios. For instance, if the filter criteria vary much and the filter criteria only remove a small number of nodes. Here, a solution could be the introduction of a parameter to give system architects the option to choose when a filter should be applied. This inclusion of the filter criteria can be integrated before the filter application, and then the filter must be added to the path finding. After checking the filter and exploitability, the algorithm creates for each vulnerability an edge. This is repeated for each neighbour and for each architectural element. In the end it then returns the attack graph as $G = (AE, E, F)$.

An example of an attack graph is given in Figure 5.13. It shows the corresponding attack graph for our running example. The nodes are the architectural elements. The edges illustrate how an attacker can compromise the connected element. Edges based on the policy are indicated by the attribute name necessary for the policy. For instance, the edge between `Terminal` and `MachineController` is labelled with `Admin` and means that the `Admin` attribute

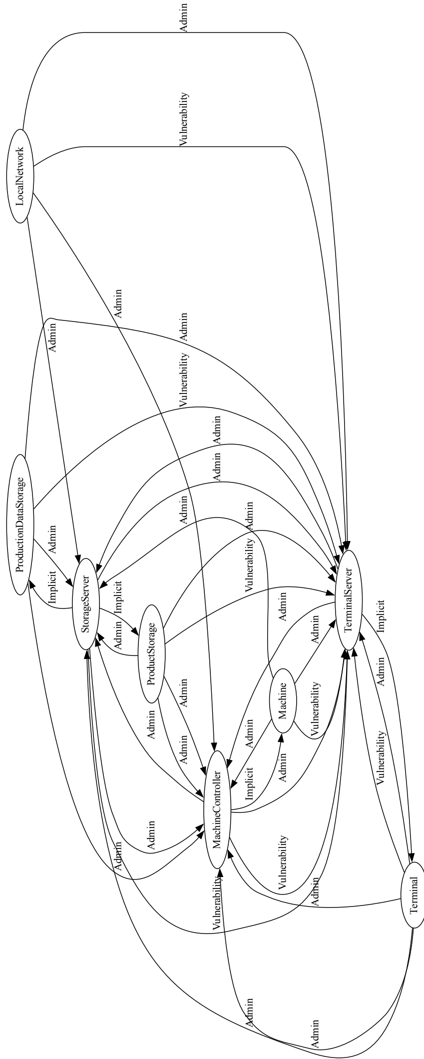


Figure 5.13.: Resulting attack graph for the running example without filters

is required. The edges which exploit the *CVE-2021-28374* [129] are labelled for simplicity reasons only with `Vulnerability` since other vulnerabilities do not exist. An exemplary edge is the edge between `LocalNetwork` and the `TerminalServer` labelled with `Vulnerability`. The last type of edges is labelled with `implicit`, indicating the exploitation of a deployment relationship. Here, an example can be the edge between the `StorageServer` and `ProductionDataStorage`.

Now to see the effects of our filtering option, we assume that the vulnerability has a `High` complexity, and we would filter for only `Low` attack complexity. This would result in the attack graph as illustrated in Figure 5.14. While in this case, it does not change the number of nodes, the number of edges is reduced. The seven edges containing the vulnerability are removed. Hence, the graph contains slightly fewer elements, which a path finding algorithm can take. This size reduction can reduce the runtime for finding an attack path.

5.3.4. Identifying Attack Paths

The previous section described the attack graph, which is the foundation for our attack path identification. For instance, Figure 5.15 highlights (grey bold, blacked dashed) two attack paths based on the unfiltered attack graph from our running example. In both cases, our target element is the `ProductStorage`, and the start point is the `Terminal`. We then gain an attack path by following the edges and searching for a path leading to the targeted element. By storing the attack path, we also get additional information about it. We get the list of involved architectural elements, vulnerabilities and access policies. These can be easily identified by using the labelled edges. For instance, with the black (dashed) path, we see that the `Admin` credential is used. Software architects or security experts can use this knowledge to break an attack path, for instance, by introducing mitigation mechanisms.

Based on the example and the description of the attack graph $G = (N, E, D)$, we can describe an attack path p from a node i to a node j as a sequence of $p_{i,j} = \langle v_i, e_1, v_{i+1}, \dots, e_l, v_j \rangle$ with $v_k \in N$ for any $k \in \{i, i+1, \dots, j\}$ and $e_l \in E$ for any $l \in \{1, \dots, l\}$ and $v_i = i$ and $v_j = j$. The set $\mathcal{P}_{i,j}$ is the set of all paths from node i to j. Hence, $p_{i,j} \in \mathcal{P}_{i,j}$. A path $p_{i,x}$ is a subpath from $p_{i,j}$ if $p_{i,j}$ is identical to $p_{i,x}$ till the element x

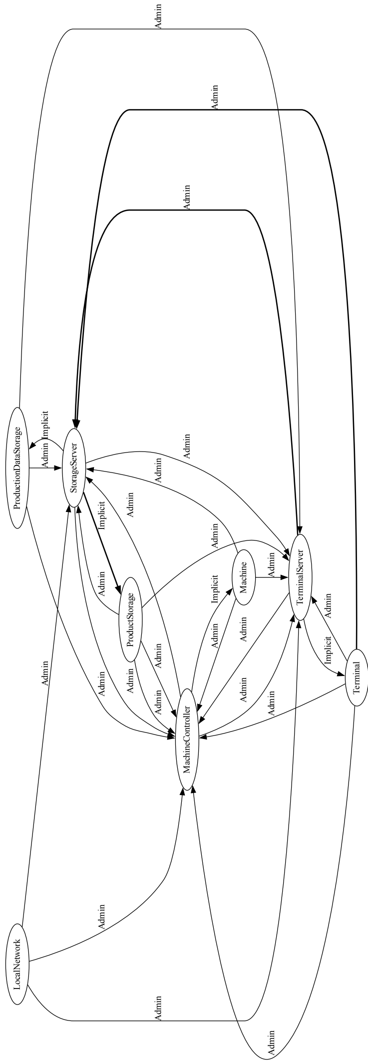


Figure 5.14.: Resulting attack graph for the running example with the vulnerability complexity filter for Low

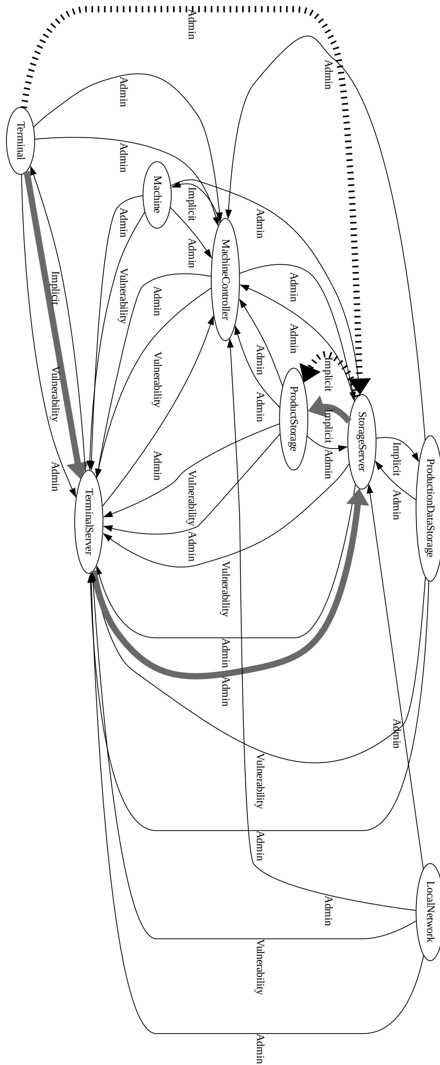


Figure 5.15.: Attack graph of the running example with identified paths from `Terminal` to the target element `ProductStorage`

For the attack path finding algorithm, we define the following helper functions:

- $filter(x, y) := x \in AE \land y \subseteq F \land \exists f \in y \mid f$ filters x
  Checks whether the architectural element x is filtered as a starting point with the filters y. This uses the `StartFilter` from Section 4.5.2.
- $length(x) : \bigcup_{i,j \in AE} \mathcal{P}_{i,j} \rightarrow \mathbb{N} :=$ number of nodes in path x
  This function calculates the length of a given attack path. The length is the number of nodes in an attack path. The number is returned as a natural number.
- $filter(x, y) := x \in \mathbb{N} \land y \subseteq F \land \exists f \in y \mid f$ filters x
  Checks whether any of the filters in y filter out the number x. This uses the `MaximumPath` filter from Section 4.5.2. In case of multiple `MaximumPath` filters, it uses the filter with the minimal length.
- $filter(x, y) := x \subseteq A \land y \subseteq F \land \exists f \in y \mid \exists a \in A \mid f$ filters y
  Checks whether any of the attributes x are filtered by any filter y
- $requires(x) : \bigcup_{i,j \in AE} \mathcal{P}_{i,j} \rightarrow ATT \subseteq A := x$ initially required attributes
  Returns a set of initially required attributes for an attack path x. Initially required credentials are these credentials, which are necessary during an attack path and cannot be provided by elements in the attack path. For instance, the black (dashed) attack in Figure 5.15 requires the `Admin` attribute. It is initially required since it is used in an edge of the attack path and no element, such as `AttributeProviders` or `vulnerabilities` in the attack paths provide it. It can be easily calculated by iterating of the attack path and comparing whether the gained attributes from the subpath till the current element already contain the credential necessary for the current edge. If not, then it is an initial required attribute.

Based on the helper functions and the previously introduced set and functions, we define our attack path finding algorithm as in Algorithm 7. The input for the algorithm is a previously described attack graph G together with the selected filter criteria F' and the selected target. The algorithm returns a set of attack paths leading to the target.

The first step in Algorithm 7 is to check whether the target is included in the graph G. If not, it returns an empty set. Afterwards, the algorithm initialises

Algorithm 7 Simplified Attack Graph Creation

Input: $G = (N, E, D), F' \subseteq F, target \in AE$
Output $PATHS \subseteq \bigcup_{i \in N} \mathcal{P}_{i,target}$

1: **procedure** ATTACKPATHFINDING
2:     **if** $target \notin N$ **then**
3:         **return** $\emptyset$
4:     **end if**
5:     $PATHS = \emptyset$
6:     $START = \{n \in N \mid \neg filter(n, F') \land n \neq target\}$
7:     **for all** $n \in START$ **do**
8:         $PATHS = PATHS \cup p \in \{p_{n,target} \in \mathcal{P}_{n,target} \mid filter(length(p_{n,target}), F') \land filter(requires(p_{n,target}), F')\}$
9:     **end for**
10: **end procedure**

the set $PATHS$ for storing the found path and determines the start nodes. Generally, every node in the attack graph that is not the target node can be a start node. In this case, the security experts or software architects get an overview of attack paths from potentially all elements leading to the target. It is only potentially for all elements since there might be elements from which no path can be found. Nevertheless, in larger systems, this result set can be very large. In the worst case, the set can contain $|N| - 1$ attack paths. Hence, it can be useful to limit the start elements in certain cases as done with our `StartFilter`. This reduces the found attack paths to the selected elements. For instance, the algorithm returns a path from any node for the attack graph in Figure 5.15. If we use a `StartFilter` with `Terminal`, it will return only one attack path from the `Terminal`, such as the black (dashed) one. This is useful as previously described (c.f. Section 4.5.2), for instance, in cases where security experts are only interested in attack paths from certain elements, such as external reachable architectural elements. The reason for this behaviour can be that the not externally reachable architectural elements are trusted more than the externally reachable ones and therefore are excluded as a potential start point.

After the identification of the start elements, the algorithm iterates over every start node and tries to find an attack path connecting the selected start node

n with the target node. We first determine all the potential attack paths between the selected element n and the $target$. From this set, we select the path matching the filter criteria. Here, it is the `MaximumPath` and the `CredentialFilter`. With the `MaximumPath` filter, we only select paths under a certain length and with the `CredentialFilter`, only paths not requiring the in the filter specified attributes. However, both can also be empty if not required. Afterwards, we take one attack path and add it to the result set $PATHS$. We choose here to only add one attack path and not all attack paths to reduce the results set. Adding all attack paths might result again in many results, which can overwhelm the analysis user. In this regard, the analysis only gives an example path similar to a counterexample that the system is not secure.

This behaviour also stresses the importance of using suitable filter criteria. An example of this importance is illustrated in the Figure 5.15 with the black (dashed) and grey (bold) attack paths. Both are valid attack paths from the `Terminal` to the `ProductStorage`. However, the grey (bold) one requires the `Admin` credential as the initial credential and the black (dashed) one does not. In general, security experts might assume that the black (dashed) one is irrelevant since the `Admin` credentials are usually better protected and not everyone might have them. Hence, the security experts could not be interested in this path and might not consider the `Terminal` as a valid starting point. However, the grey (bold) path does not initially require the `Admin` credential. It still requires the `Admin` credential for the compromisation step from the `TerminalServer` to the `StorageServer`. In contrast to the black (dashed) attack path, the grey (bold) one gets this attribute by exploiting the vulnerability from the `TerminalServer`. This exploitation step is prior to the step where the credential is needed. Hence, the attacker can use it in the later step, and it is not initially required for this path. The selection of the grey (bold) attack path can be forced by using the `CredentialFilter` and selecting the `Admin` as forbidden initial credentials.

Remark: The actual implementation of the attack path finding algorithm varies in the behaviour a bit to the described algorithm. In our implementation, we choose to limit the path finding to the identification of simple paths. A simple path is a path where the path $p_{i,j}$ only contains unique elements. In other words, it contains no loops of any kind, and no element exists more than once. In addition, we use an implementation based on Yen's ranking loopless path algorithm [108] for the path finding. This has two reasons. First, by not limiting the paths to simple paths, the execution time for identifying a path increases drastically in the underlying graph framework. Hence, even for

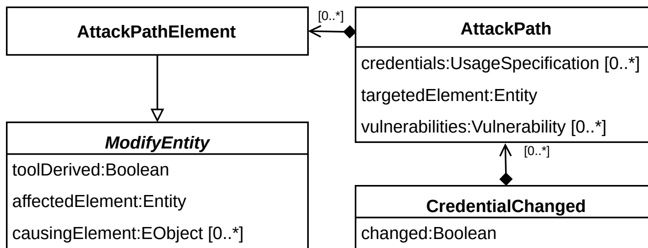


Figure 5.16.: Metamodel elements for the result of the targeted attack graph analysis

small systems, the execution time is unsuitable. The reason is that the graph framework can only identify all the paths and based on our graph definition, the graph is not acyclic. Therefore, theoretically, there can be infinite paths. Here, we would need some heuristic approaches, which decide when to cut the search or loops. For example„ the graph framework allows setting the maximum path length and then cutting the search. Yet, this still comes with a high computational effort. In contrast, our chosen algorithm can calculate the path in $O(kn(m + nlogn))$ with n as the number of vertices and m as the number of edges and k as the number of paths required. The second reason is that the selected path algorithm is the only one in the used graph framework, which allows us to add a custom filter for the path finding. Therefore, only this algorithm in the used framework allowed the implementation of the `CredentialFilter`. Unfortunately, this implementation choice comes with additional drawbacks, especially regarding the accuracy. We will discuss them in our evaluation in Section 7.3.

5.3.5. Result Model for the Targeted Attack Graph Analysis

The result of the analysis is a list of attack paths. Similar to the attack propagation analysis, the result is stored in an EMF result model. The basic structure is that a list of affected elements exists for each path.

The result metamodel is illustrated in Figure 5.16. It contains two elements already introduced in Section 5.2.5: The abstract `ModifyEntity` and `Credential-Change`. The `CredentialChange` is the container for the `AttackPath` elements. An `AttackPath` encapsulates an attack path and its description. As attributes, it has `credentials`, which are the `UsageSpecifications` used in an attack

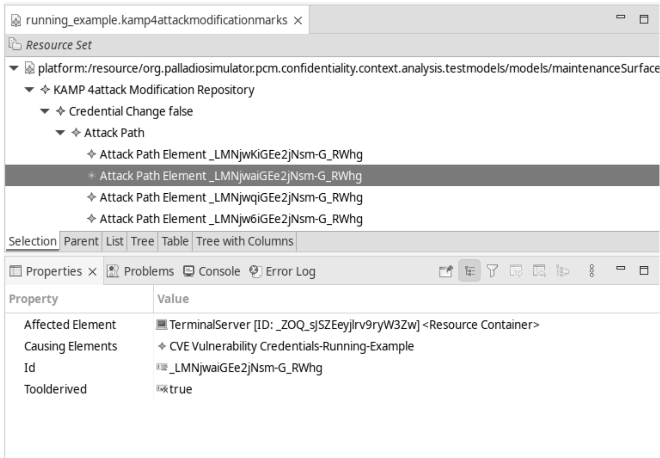


Figure 5.17.: Attack path result example for the running example in *Eclipse*

path. In addition, it has the `targetedElement` and a list of `vulnerabilities` with the used `Vulnerability` elements. Each `AttackPath` also contains a list of `AttackPathElements`. These represent the actual compromised architectural elements.

Figure 5.17 show an exemplary output in our tooling. For this result, we use the running example with a `CredentialFilter` with the `Admin` and a `StartFilter` selecting the `Terminal`. The upper half of the figure shows an excerpt of the tree editor with a model instance of the result metamodel. It shows how the `AttackPath` is contained by the `CredentialChange`. The `AttackPath` then contains the affected architectural elements in this attack path. The lower half shows more detailed information about the selected `AttackPathElement`. The detailed information is similar to the attack propagation. It shows the affected element, in this case, the `TerminalServer`, the reason for the compromisation, here the exploited vulnerability, the id, and whether it was derived from the analysis or not. A slight difference in the output is that it does not show the origin of the attack. However, this is unnecessary since the origin is always the previous element in the list. This is based on the fact that the list is ordered from the start element to the target element and the analysis can only detect simple paths. A simple path has no loops so that the previous element is always the origin of the attack.

Based on this result model, the software architects can then evaluate the software architecture. They have to decide together with a security expert whether the architecture with the existing attack paths is suitable or whether they have to mitigate some attack paths. The mitigation of the attack paths is similar to the previous attack propagation. They can, for instance, add mitigation concepts or change the used credentials.

Part III.

Validation

6. Evaluation Scenarios

In the previous sections, we answered our research questions by introducing our contributions. We answered **RQ1** by introducing our access control metamodel together with our scenario analysis and **RQ2** by introducing the access control, vulnerability, attack and attacker metamodel and combining it with the attack analyses. In this section, we present the different evaluation scenarios which we use to evaluate our contributions.

For the evaluation of our approach, we use different evaluation scenarios. These scenarios are based on various different sources, such as evaluation case studies from related approaches or public descriptions of security incidents. For us, an evaluation scenario describes a system and security situations for the system. Examples of security situations are, for instance, the access of the external technician to the machine in our running example or the described attack propagation in our running examples. Besides the system description and security situations, our evaluation scenarios are exploratory in the sense that we want to investigate how our approach behaves in certain situations. Additionally, each scenario has a defined objective, which is to answer our evaluation questions. This brings the benefit that a scenario-based evaluation can provide insights into the application of our analyses and indicates the applicability of our approach. It can also increase the comparability to other approaches using the same scenario and give insights into how our approaches might behave in different scenarios with similar properties. These described properties are also properties of a case study as defined by Runeson et al. [157]. However, the main distinction is that we do not study "the phenomena [...] in their normal context" [157, p. 29]. The normal context is the application of our approach by software architects and security experts during the regular work to design a system. In our evaluation, we only simulate this behaviour by performing similar steps as software architects and security experts. As a result, we refer to them as evaluation scenarios rather than case studies.

Table 6.1 shows important characteristics for each scenario. The first column contains the scenario name. Afterwards, it lists the number of components,

hardware resources and network resources for each scenario. The next group illustrates for which contribution the evaluation scenario is used. A cell marked with *x* indicates the application and a dash (—) indicates no application. The first column in this group is the contribution C3 with the *Scenario-Based Access Usage Analysis* (c.f. Section 5.1), the second column in this group is the contribution C4.1 with the *Attack Propagation Analysis* (c.f. Section 5.2), and the third column in this group is the contribution C4.2 with the *Targeted Attack Graph Analysis* (c.f. Section 5.3). The last group shows some more insights into the expected results or specific properties of the analysis. If there is no additional information, the cell contains a dash (—). The first column in this group is the number of misusage and usage scenarios. The second column contains the number of affected elements in an attack propagation, and the last column the number of attack paths.

Each scenario description is structured the following. We will first explain the source of the scenario. Afterwards, we describe the general setting of the scenario. Then we will provide an architectural overview and description of the scenario. This also includes an overview of the software architecture. For the overview, we follow the syntax and semantics which we introduced for the running example in Chapter 3. After the architecture, we will describe the access control properties and vulnerability properties. Then we describe the expected output and possible variations used in the different analyses for each scenario.

6.1. TravelPlanner

Source The TravelPlanner scenario originates from a case study in the iFlow project [91]. Katkalov [90] use it as an evaluation case for evaluating his information flow analysis. Kramer et al. [99] developed based on the available description a PCM model and used it in the evaluation of their attacker analysis. The scenario is also used in the evaluation and as a showcase scenario in the data-flow-based PCM confidentiality analysis [67, 166, 167, 168]. There are also some extensions regarding the confidentiality analysis under uncertainty with the approaches Walter et al. [210] and Hahner et al. [65].

For our approach, we used the model from Kramer et al. [99] and extended it with our security properties. We also used this scenario in our publications

regarding the attack propagation [211], the access analysis [214], the attack path analysis [212], and in another publication where we coupled our attack propagation (C4.1) with a data flow analysis to estimate the criticality of data [209].

Description The scenario describes a simple flight booking system. It consists of three different entities: The *User*, *Travel Agency*, and *Airline*. The user can, via a mobile app, search for flights at the travel agency. The travel agency gets the flight from the airline. After selecting a flight, the user can book a flight at the airline. The flight is paid for by a credit card. The credit card data is only released to the airline if the user explicitly declassifies it. After the booking, the airline pays the travel agency a commission. The important aspect is the declassification of the credit card data. Without an explicit declassification, the credit card should not be released. As a further addition, we added the constraint that the declassification (explicit release) should only be possible at the home location of a user. This constraint illustrates the context-based access decision of our approach.

Architectural Overview As already described, we reused the architectural model from Kramer et al. [99]. For the evaluation, we removed their security annotation for physical attack propagation since we do not require them and there were technical issues with the annotation process. Otherwise, we reused their architecture. An architecture overview is given in Figure 6.1. Each entity has a hardware device, the *user* a `MobilePhone`, the *airline* the `AirlineServer` and the *agency* the `AgencyServer`. These hardware resources are connected by multiple network resources. In the original model, they indicated different physical attacks on the network device. In our case, they represent different network layers. The behaviour of the *airline* and *agency* are encapsulated in a component for each entity. These provide different services. For instance, the `Agency` provides a service for paying the commission, which is required by the `Airline`. The user behaviour is separated into two components. The first is the `TravelPlanner`. This covers the flight selection and booking of the user. The payment process is modelled within the `CreditCardCenter`. This component provides services to declassify the credit card details as required by the initial description and afterwards services to release the credit card data.

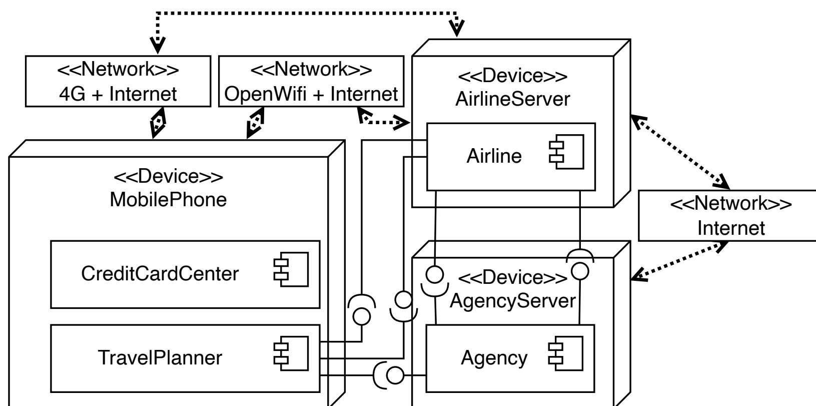


Figure 6.1.: Architectural overview TravelPlanner scenario

We manually derived the access control policies for the services based on the description of the scenario and the modelled security properties by Seifermann et al. [167] and Kramer et al. [99]. They can be roughly described as each entity can access the services needed for their behaviour. This basically breaks down in a policy model similar to RBAC. Each entity is assigned a role and this role is granted access. However, for some services, this access control modelling is insufficient and handling of dynamic attributes is required. These are the services for the `CreditCardCenter`. The first service is the declassification. The policy requires that the user needs to be at home for the declassification. We modelled this by introducing an attribute representing the user's location and added the home location to the required attribute values. The second service is the release of the credit card data. This should be only possible if the data is previously declassified. In our case, we modelled this by adding an attribute for the declassification and considering it in the access control policies. Unfortunately, there was no access information regarding the hardware devices available. However, these are important for our approach. Therefore, we decided to extend the model with our own access control policies, which we describe in the following. We defined for the *airline* and *agency*, each an admin role. Each admin role has access to the respective server it manages, meaning the *airline* admin has access to the `AirlineServer` and the *agency* admin has access to the `AgencyServer`. We assume that this is a reasonable extension since, in companies, one admin is

usually responsible for the servers. For the `MobilePhone`, we assigned that the user has access to it since, for a private entity, usually, the user has admin control on their devices.

The scenario is based on an evaluation case in the confidentiality and information flow domain. Hence, the description and the modelled variants contain information regarding the access to data or services. The existing information does not contain any information regarding the vulnerability of the components or hardware resources. However, for evaluating our attack analyses, it would be beneficial to have also this information. Without access information, our evaluation would only be able to cover the compromise of the system based on access control policies, not on vulnerabilities. Therefore, we decided to extend the scenario with vulnerabilities manually. Since we wanted the added vulnerabilities to be representative for these kinds of systems, we choose to research common security problems. We limited our search to CWE vulnerabilities since we had no concrete implementation from which we could extract the concrete CVEs. In our case, we extracted the vulnerabilities from the OWASP 10 [137], which lists the most common security threats in web applications. In our case, the scenario is not a web application but the used technologies and concepts to realize the scenario are similar. Therefore, we can assume that they can contain the same security vulnerability. Each OWASP security threat also contains a detailed description. These descriptions also contain reference CWEs representing the vulnerability. In our case, we choose instances of these for our vulnerability modelling. Therefore, we assume that the vulnerabilities are representative.

Usage and Misuage Scenarios Our scenario contains two `UsageScenarios` and one `MisusageScenario`. The first `UsageScenario` represents a regular user interaction with the system. This scenario covers the search for a flight, the booking, the declassification of the credit card and the payment of the commission. The scenario is based on the related approaches, which consider the same scenario. The analysis should consider this as passed. The second `UsageScenario` describes the setting of the credit card. This scenario is also based on related approaches [166]. This scenario should pass. The `MisusageScenario` models the same scenario. However, the attributes are wrong and do not match the required attributes. Therefore, the access control decision should be denied, which leads to a pass since it is a misusage scenario. These tree usage descriptions already contain some of the important aspects of our

access analysis. It has denied and permitted access decisions. In addition, it uses `UsageScenarios` and `MisusageScenarios`.

Attack Propagation For the attack propagation, we created 15 subscenarios for the TravelPlanner. Each represents important propagation conditions for our analysis. We used here the subscenarios since some edge cases for the propagation were not considered in the other scenarios, such as that no propagation is possible. Each subscenario uses the same architectural model, but the attacker, vulnerability, or access control model can vary slightly. Therefore, also the number of affected elements varies between the subscenarios. Depending on the subscenario, between 0 and 12 elements are affected. Our dataset [208] contains a complete overview of the subscenarios.

Attack Graph & Paths In this scenario, we are interested in attack paths leading to the `CreditCardCenter` without using the `User` attribute. We choose this attack path because, in this scenario, the critical confidential data is the credit card data. The credit card data is managed in our case by the `CreditCardCenter`. In addition, the `CreditCardCenter` is not directly connected to other external components. Hence, it is an interesting case to investigate how it can be compromised. The usage of the filters excludes the obvious solution of using the credential of the `User` to compromise the `CreditCardCenter` directly. We exclude this path because, similar to the root credentials, we assume that an attacker does not have the user privileges. In our case, this should lead to one attack path from the `MobilePhone`. This is the only attack path since otherwise there is no possibility to gain the `User` attribute and the `CreditCardCenter` has no vulnerability to exploit. This scenario already covers attack path finding with implicit take-overs and the usage of filter attributes.

6.2. Power Grid

Source For the *Power Grid* scenario existed no architectural model for PCM. However, two reports describe the incident [68] [170]. Both contain information about the used vulnerability and attack walkthroughs. The first also contains a basic overview of the involved network elements.

We also used this scenario in our publications regarding the attack propagation [211], the attack path analysis [212] and additionally in Walter et al. [209].

Description The scenario is the cyberattack on the Ukrainian power grid at the end of 2015. During the incident, malicious users entered the business backend of a power supplier by using compromised credentials. Afterwards, the attacker propagated through the network and gained control of various services, such as the domain controller or the call centres. The business backend was connected by VPN to the Industrial Control Systems (ICS) network. During their propagation, the attacker also got access to credentials for the VPN to the ICS network. The ICS network contained *DMS* servers and clients for managing the breakers of the power grid. Breakers are devices to disconnect a power grid. The attacker gained access to them and opened the breakers, which led to a power outage. In addition, they disabled the call centres and the network to increase the damage and lengthen the repair time [68]. In our scenario, we only investigate the propagation part, i.e., after the initial compromisation and till the propagation ends with accessing the breakers.

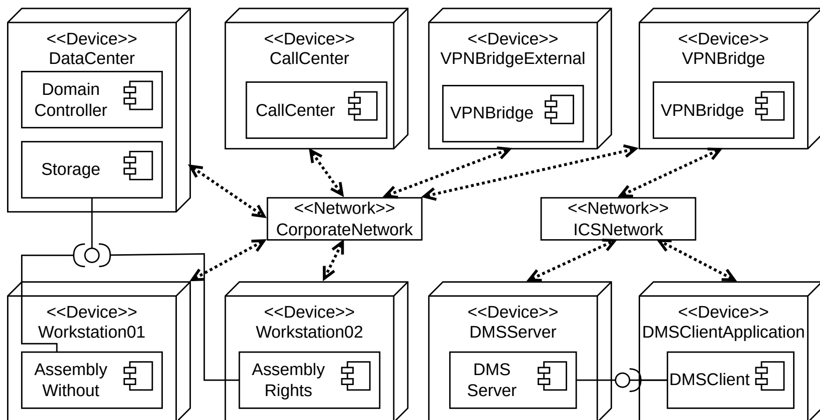


Figure 6.2.: Architectural overview *Power Grid* scenario

Architectural Overview Since there was no PCM architecture available, we created the architecture based on the existing reports describing the system structure [68] [170]. Most of the elements involved are inspired by the attack walk through and the shown network schematics in [68, p. 12]. However, the report does not contain a complete software architecture but more like a structuring of the involved network topology and how they were affected. Hence, we needed to extend the given network topology with components and services. Our scenario system consists of nine components, eight hardware resources and two network elements. The main part in this scenario is the usage of network layers separating the business backend from the ICS. In our scenario, this is modelled by the `CorporateNetwork` for the business backend and the `ICSNetwork` for the ICS. These networks are connected by the hardware resource `VPNBridge` representing the VPN in the original description. The `CorporateNetwork` contains different `WorkstatationX` on which components are deployed. These represent the different office computers from employees. These components are connected with a central `Storage` component, deployed on the `DataCenter`. On the `DataCenter` also the `DomainController` is deployed. In addition, the network contains a `CallCenter` device with a component and a device representing an external VPN node. In the `ICSNetwork`, there are the devices `DMSServer` and `DMSClient` with their respecting components. These represent the devices and services for controlling the breakers. The architectural model simplifies the software architecture since not all provided and required connectors are explicitly modelled and the office devices contain only one component and not multiple ones. However, already this basic model represents all the required elements identified in the literature. In addition, it contains interesting aspects for an attack propagation, such as the propagation between network layers or the gaining of attributes.

This gaining of attributes is illustrated with the access control model. The original attacker had no knowledge about the necessary credential to access the VPN to the ICS but gained it during the propagation. We model this gaining by assigning `AttributeProviders` to the `AssemblyContext` `AssemblyRights`, indicating that there the attributes for accessing the VPN are stored. In addition, we assign an `AttributeProvider` for the `DomainController`, which provides the attribute `BackOfficeAdmin`. This attribute grants access to all the devices in the `CorporateNetwork`, excluding only the `VPNBridge` to the `ICSNetwork`. For accessing the `VPNBridge`, the attributes for the VPN and user rights for the ICS network are necessary. The other elements in the ICS only require the user rights.

Regarding the vulnerabilities, we annotated CVE-2014-1761 [126] to the `AssemblyRight`. It is a vulnerability for Microsoft Word and was also used by the worm (BlackEnergy), which was used by the attackers. Therefore, it should be representative.

Usage and Misuage Scenarios We do not use this scenario in the access analysis since the description does not contain enough information about the behaviour of services.

Attack Propagation The expected attack propagation is based on the attack walk through from Hamilton [68]. We choose to use the attack walk through as a foundation for the attack propagation because it provides an external reference for the propagation. Also, it represents an actual attack on a system. In our case, we choose the `Workstation01` as a starting point for the propagation. This differs from the originally described attack because there the starting point is an initial phishing mail attempt to get credentials to a workstation. For our analysis, we excluded the consideration of social engineering attacks to which phishing attacks belong. However, we still can use the described attack because we can derive an initial starting point from the attack. For deriving the starting point, we select the propagation after the initial access by the phishing attack is gained. This means we can reuse the described scenario. In our case, the start element is the `Workstation01`. Besides the starting point of the attacker, the attacker also needs capabilities. The capabilities are derived from the modelled vulnerabilities, which are based on the description of the used vulnerabilities. The affected elements of the propagation are also based on the report. The report mentions that multiple elements, such as various workstations, servers and ultimately the services to manage the breakers are affected. In our modelled case, this means that the propagation needs to propagate from the `CorporateNetwork` to the `ICSNetwork`. There, it can reach the services provided by the `DMServer` and `DMSClient`. In our scenario, these represent the breakers, which were the goal of the originally described attack.

Attack Graph & Paths Similar to the previous attack propagation, we exclude for the attack paths the initial social engineering attack. The targeted element is the `DMSClient` since these are responsible for the breakers in our scenario. These breakers would give attackers the capability to switch the power grid

off. In the original attack, this was the main goal of the attacker. Therefore, we also choose this as a target. Other potentially interesting targets exist, such as the `CallCenter` or `Domain Controller`. However, in our case, we focused on the architectural element, which potentially can generate the most damage. These are the breakers because of their capability to turn off the power. Besides the target definition, we also used a `CredentialFilter`. The filter contains all the used attributes in the system. This represents an attacker with no initial knowledge about credentials. This forces the analysis to choose more complicated attack paths because otherwise, the attacker could use credentials and does not need to identify vulnerabilities to get more privileges. This is consistent with the attack report [68], which state that the attackers gained further access to the systems during the propagation. For our evaluation scenario, the start element can be any element in the architecture. We did not further restrict the analysis here because all elements within can be reasonable start points. Considering all architectural elements as a starting point would lead to 19 attack paths. However, because of our filter criteria and the modelled software architecture, three of them cannot reach the targeted element (`ICSNetwork`, `DMSServer` (device), `DMServer` (component)). This leads to 16 expected attack paths.

6.3. Target

Source There exists no publicly available complete software architecture. However, different reports [173] [145] and expert web entries[1] state the involved components and discuss attack scenarios.

We also used this scenario in our publications regarding the attack propagation [211], the attack path analysis [212] and additionally in Walter et al. [209].

Description This scenario is based on the Target breach in 2013. Target is an American retail chain. Its business backend was allegedly compromised by a compromised external supplier [173], who had access to the business

[1] B. Krebs. *Inside Target Corp., Days After 2013 Breach.* Sept. 21, 2015. URL: `https://krebsonsecurity.com/2015/09/inside-target-corp-days-after-2013-breach/` (visited on 09/01/2021).

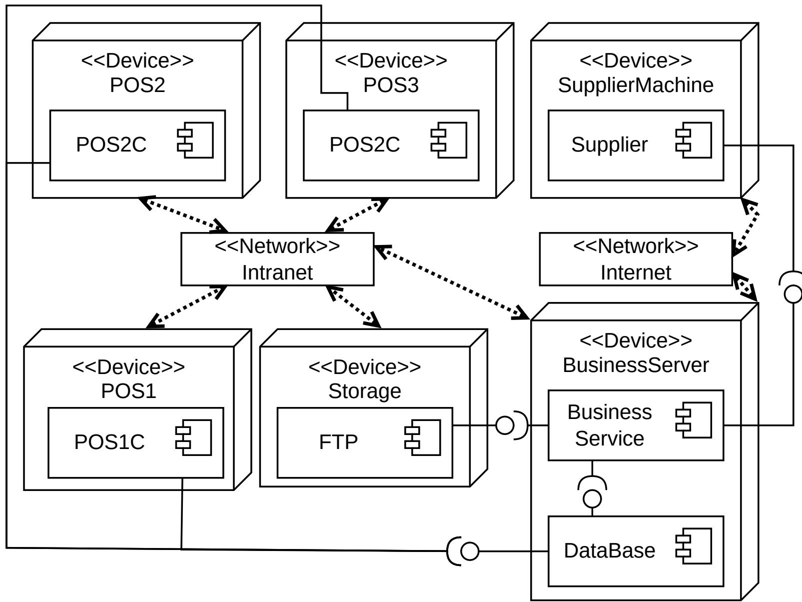


Figure 6.3.: Architectural overview *Target* scenario

backend with the billing services. The attacker exploited there some privilege escalation vulnerabilities to get more privileges [173]. This resulted in the compromisation of various different services and entities, such as the billing service, FTP-storage servers, databases and the Point of Sale (POS) devices. The last ones are the devices where customers can pay by using their credit card. Hence, also the customer credit card data was affected since they were unencrypted. Krebs[2] states that often weak passwords or default passwords were used. Also, the used software was outdated and they gained access to a domain admin account [145].

[2] B. Krebs. *Inside Target Corp., Days After 2013 Breach.* Sept. 21, 2015. URL: `https://krebsonsecurity.com/2015/09/inside-target-corp-days-after-2013-breach/` (visited on 09/01/2021).

Architectural Overview We extracted the architecture based on the reports mentioned in the source description. A simplified overview is given in Figure 6.3. Overall the architecture contains 7 components, 6 devices and two network nodes. The scenario uses again the concept of network segregation to differentiate between the external network and the internal network because also the real Target network probably used network segregation. In our model, this is realized by the network elements `Intranet` and `Internet`. The first is the Target internal network. The external compromised supplier is modelled as the device `SupplierMachine` on which the component `Supplier` is deployed. This component is connected to the billing service provided by the `BusinessService` which is deployed on the `BusinessServer`. This is also the gateway to the `Intranet`. On the `BusinessServer` a `Database` is deployed. It is connected to the `BusinessService` and the POS components. The `BusinessService` is connected to an `FTP` component deployed on the `Storage` device for storing large objects. In our scenario, we modelled three POS devices. Of course, in reality, there might be much more, but already the three ones are enough to evaluate the basic functionality. Each POS has its own device, where its own component is deployed. They contain services which handle the credit card data.

Our scenario contains two roles. The first is the domain admin with access to all Target devices, including the `Intranet` device. The second is the supplier with access to the billing service. While this access model looks simple, it contains the important aspects of the described scenario. First, the supplier who has only access to the billing service. Second, the domain admin, who was somehow compromised. We also added `AttributeProviders` for gaining the supplier role and the domain admin.

For the annotated vulnerabilities, we use the mentioned weakness and annotated architectural elements with `CWEVulnerability` elements. The annotated elements are the billing service and `BusinessService` with a vulnerability for privilege escalation, the POS devices with weak passwords, and `FTP` with default passwords.

Usage and Misuage Scenarios We do not use this scenario in the access analysis since the description does not contain enough information about the behaviour of services.

Attack Propagation The attack propagation is based on the described incident reports because by using the incident reports, our attack propagation is based on a real security incident. These reports name that the attack originated from the supplier system. Therefore, we select for the attacker as a starting point the `Supplier`. In addition, Shu et al. [173] state that the attacker needed capabilities to exploit a vulnerability in the business backend of *Target*. Therefore, we give the attacker the capability to exploit the mentioned privilege escalation vulnerability in the `BusinessService`. In addition, we gave the capabilities to exploit the other vulnerabilities because Plachkinova et al. [145] mentions a security report containing these vulnerabilities. In the original attack, the attackers compromised the credit card data of customers by compromising the POS devices. Hence, the attack propagated from the *Supplier* over the *Business Backend* to the POS device. Thereby, they compromised different elements. In our case, we represent this behaviour by expecting that all the architectural elements within the `Intranet` plus the start point are compromised. Therefore, we assume that our expected output is representative for the Target breach.

Attack Graph & Paths Here, our targeted element is the `POS1C`. The scenario is, that the security experts are interested in whether attackers could somehow get to a POS component and see the confidential credit card data. The scenario resembles the real-world breach, where the attackers stole the credential card data. Hence, security experts can prevent similar attacks by trying to mitigate these found attack paths. As a filter, we selected the credential filter with the domain admin attribute since attackers usually do not have initially some kind of admin credentials. Also, in the originally described scenario, the attacker presumably had no access to the domain admin at the beginning. Therefore, we assume that the usage of this filter is justified. In contrast to the originally described breach, we select any architectural element as start points for the attack. We decided to investigate all the possible attack paths here because we are interested in finding many attack paths to investigate how our analysis performs for different path types. In our case, without the targeted element, these are 14 elements. For all these elements, it is possible to create an attack path to the targeted element. Hence, we expect 14 attack paths. These different attack paths include already different kinds of attack propagations, such as overcoming network segregation or gaining credentials to compromise other components.

6.4. Cloud Infrastructure

Source This scenario is based on the description of a cloud scenario in Alhebaishi et al. [10]. It describes not an existing breach or system, but according to the authors, it is based on concepts and ideas from real-world products in the cloud domain. We used the *cloud data center infrastructure 1* in our scenario.

We also used this scenario in our publication regarding the attack path analysis [212].

Description The scenario describes a simplified cloud infrastructure. Containing different network layers and components found in reference scenarios.

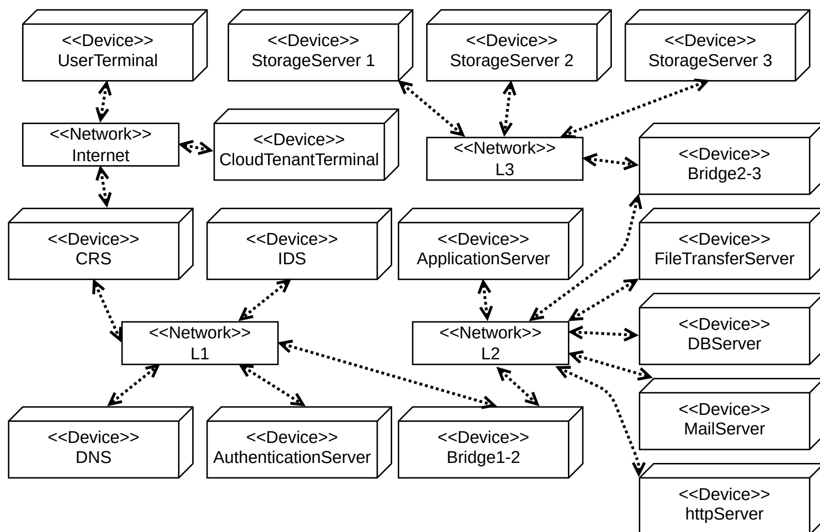


Figure 6.4.: Hardware resources and networks in the cloud infrastructure scenario

Architectural Overview We extracted the architecture based on the description in Alhebaishi et al. [10]. However, the description is more like a network topology. Hence, our architecture lacks the more fine-grained architectural

description with the required and provided components. Nevertheless, this scenario is interesting since it shows how our analyses behave with this kind of abstraction. Figure 6.4 illustrates the scenario. For a better overview, we leave out the components in the figure since they do not bring many insights into this scenario. The figure only contains the networks and hardware resources. The hardware resources are connected to the network by dashed lines indicating that all the hardware resources within a network can communicate with each other. Overall the scenario has 11 components, 16 hardware resources and four networks. The four networks are: The `Internet`, `L1`, `L2`, and `L3`. They represent different network zones. They are connected by different devices acting as routers or bridges. For the naming, we followed the original description. Hence the bridging component between `Internet` and `L1` is called `CRS`, which stands for a router from Cisco. In addition, the `Internet` contains the devices `UserTerminal` and `CloudTenantTerminal`. The `L1` contains beside the bridges an `IDS`, a `DNS`, and an `AuthenticationServer`. The `L2` contains in addition to the bridges a `FileTransferServer`, a `DBServer`, a `MailServer`, and a `httpServer`. The third layer then has three devices for storage.

The access control and vulnerabilities are extracted from the scenario description. The access control policies assign the access to different roles and the vulnerabilities are assigned to different devices.

Usage and Misuage Scenarios We do not use this scenario in the access analysis since the description does not contain enough information about the behaviour of services.

Attack Propagation We do not use the attack propagation in this scenario since the description is more about attack paths.

Attack Graph & Paths The targeted element is, in this scenario, a component deployed on the `DBServer`. This target definition is derived from the original scenario description [10]. It represents a hypervisor component. In this scenario, we do not use any filters because no restrictions are named in the original description. We expect, in this scenario, 14 attack paths.

6.5. ABAC-Banking

Source The system is based on an evaluation case in Seifermann et al. [168] and Seifermann [166].

We also used this scenario in our publication regarding the access analysis [214].

Description The scenario describes a data-flow-based ABAC banking system. Different branches with different customer types (celebrity and regular) exist in it. Based on the customer type, either a manager needs to handle the requests or a regular clerk. In addition, the access is restricted to the locations. For instance, a clerk in Asia cannot access the American banking operations. Besides the handling of accounts, regular customer can also transfer their accounts between different branches.

Architectural Overview The architecture is based on the PCM models provided by Seifermann [166]. It consists of 11 components, from which two are `CompositeComponents`. In addition, it has two hardware resources and one network node. Figure 6.5 illustrates the components' architecture. We leave out the deployment since, in this scenario, it is not relevant. The different branches are realized by two `CompositeComponents`, the `BranchOffice-USA` and `BranchOfficeAsia`. Each component contains a `CustomerStore` and `CustomerHandlingRegular` for handling and storing regular customers. The same structures exist for celebrities. Between the different branch components is the `CustomerMovement` to transfer customers between branches.

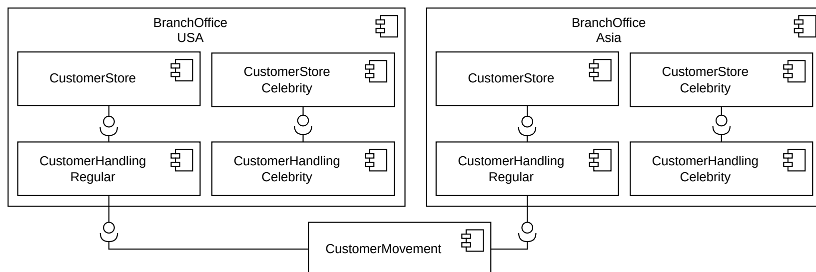


Figure 6.5.: Simplified overview ABAC banking scenario

The access control properties are derived from the original ABAC model. However, we needed to adapt them from a data-flow-based definition to our service-based definition.

Usage and Misuage Scenarios We investigate in this scenario two usage scenarios and one `MisusageScenario`. The usage scenarios are extracted from the original description of the scenario [166]. They represent two typical cases in the system. One usage scenario covers the handling of a regular customer by a clerk and the other covers the handling of a celebrity by a manager. Here, all the access decisions should be *permit* and the scenarios should be passed. The misusage scenario is about the handling of a celebrity by a clerk. We derived it from the description, where potential error cases are described. These error cases can be considered as misusage. Hence, we derived a misusage scenario from them. In our modelled misusage scenario, the access decision should be *deny*, and the scenario should be passed.

Attack Propagation and Attack Graph & Paths We do not use this scenario with our attack analyses since the used PCM model uses `CompositeComponents` and they are not compatible with our attack analyses. Despite that limitation, we choose this PCM model since it is used in related work.

6.6. Education

Source This scenario is based on a described example system in Fisler et al. [57]. The authors state that the described system is based loosely on the real policies of the authors' university. Based on the description, we extracted a software architecture and access control policies.

We also used this scenario in our publication regarding the access analysis [214].

Description The scenario describes a grade management system at a university. It needs to consider two aspects, first, the operations of assigning, reviewing, and receiving grades. Furthermore, there exists the concept of external and internal grades. The operations are performed by different roles, such as teaching assistants or faculty, on the grade types. The scenario also

describes violations, such as that a teaching assistant cannot assign external grades.

Architectural Overview The architecture consists of two instances of a grade management component. The grade management component provides three services providing the assign, receive, and view functionality of the system. We instantiate it once for the internal grade and once for the external grade. These components represent the concept of internal and external grades. Both instances are deployed on the same hardware resource and connected to the same network. However, the deployment is done only for a complete PCM model since the original scenario does not describe it or consider it. It also does not influence our analysis results in this scenario.

The access control policies are extracted from the dataset of [57]. The dataset contains XACML files with different access control rules. However, we cannot use them directly since they are written for a previous XACML version. Hence, we needed to update them. In addition, we needed to add the references to the PCM architecture.

Usage and Misuage Scenarios We investigate two usage scenarios and two misusage scenarios. The usage scenarios are modelled along the described usage found in Fisler et al. [57]. One usage scenario is about the assignment and viewing of internal grades by a teaching assistant. The other one is about the assignment and viewing of internal and external grades by the faculty. For both scenarios, the expected access decisions are permit. Hence, the expected output for the scenario is passed. The misusage scenarios are derived based on the violations and problems. The first misusage scenario is the assignment of external grades by a student. The expected decision is deny and then the misusage scenario should be passed. The second misusage scenario describes that a teaching assistant assigns an external grade. As already stated in the description, this should not be possible. Hence, the expected output for the decision is deny to pass the misusage scenario. An expected failed misusage scenario is the assignment of an external grade by the faculty since this is allowed. The failed usage scenario is the assignment of an external grade by a user without a role.

Attack Propagation and Attack Graph & Paths We do not use the attack analyses with this scenario since the described scenario primarily focuses on access control decisions and does not consider vulnerabilities or attackers.

6.7. Maintenance Scenario

Source/Description/Architectural Overview This scenario is based on our running example (c.f. Chapter 3).

We also used this scenario in our publications regarding the attack propagation [211], the access analysis [214], the attack path analysis [212], and in Walter et al. [209].

Usage and Misuage Scenarios We derived four usage and two misusage scenarios from the scenario and architecture description. The first usage scenario is the regular behaviour of a technician accessing the `Terminal` during a failure state. The second usage scenario is the `Machine` saving its log data. The third usage scenario is the access of a *Product Developer* to the `ProductStorage`. All these use cases should be passed. The fourth usage scenario should fail because it tries to store data without authorisation. The misusage scenario is identical to the first usage scenario, with the only difference being that the machine failure attribute is missing. Hence, the access control decision should be denied for the misusage scenario to pass. The last misusage scenario is identical to the first usage scenario except that it is a misusage scenario in contrast to a usage scenario. Therefore, it should fail.

Attack Propagation The investigated attack propagation, is similar to the described example in Section 5.2. The attacker starts at the external `Terminal` and has the capabilities to exploit *CWE-312*. In the end, we expect a propagation to the `StorageServer` and its components over the `TerminalServer`.

Attack Graph & Paths The investigated attack path is similar to the last path in Section 5.2. The attacker has a `CredentialFilter` for the admin attribute. The targeted element is the `ProductStorage`. Based on this configuration, we expect seven attack paths.

Remark: In contrast, to the described running example in the previous section, we do not use a `StartFilter` selecting the `Terminal`. During the evaluation, we saw that restricting the starting point could influence the results. By using the filter, we achieved better results regarding the accuracy. Therefore, we choose not to use the `StartFilter` here.

Table 6.1.: Characteristics of the evaluation scenarios

Evaluation Scenario	Component	Hardware	Network	C3	C4.1	C4.2	Mis-/Usage	Affect.	Paths
Target	7	6	2	—	x	x	—	22	14
Power Grid	9	8	2	—	x	x	—	27	16
Cloud Infrastructure	11	16	4	—	—	x	—	—	14
TravelPlanner	4	3	3	x	x	x	3	0-12	1
ABAC-Banking	11	2	1	x	—	—	3	—	—
Education	2	1	1	x	—	—	6	—	—
Maintenance	4	3	1	x	x	x	6	14	7

7. Evaluation

In the previous section (c.f. Chapter 6), we presented different evaluation scenarios. These evaluation scenarios are different cases, either inspired by real-world system breaches or confidentiality evaluation cases, and we described the source of the scenarios and important properties. In this section, we use these scenarios to evaluate how well we answer the research questions with our contributions.

The Sections 7.1, 7.2, and 7.3 describe the evaluation for each analysis. We choose to only directly evaluate our analysis contributions (C3, C4) and not the metamodels contributions (C1, C2). However, the metamodels are indirectly evaluated based on the introduced model definition by Stachowiak [187] in Chapter 2. As previously described, each model needs to serve a pragmatism. In our cases, this pragmatism is the individual analysis. Hence, if the metamodels fail their pragmatism, the analyses should be affected. Based on this relationship, the contributions C1, and C2 are indirectly evaluated based on the evaluation of the contributions C3, C4.1, and C4.2.

Each analysis evaluation follows the Goal Question Metric (GQM) [19, 20] approach. This approach breaks down the evaluation task into multiple evaluation goals. These evaluation goals are then further specified by evaluation questions, which are answered by metrics. This helps to align the used metrics to the evaluation goals and makes clear why the metric is used in the evaluation. In addition, a well-defined GQM plan can help to increase the reproducibility since other researchers have a structure to follow. Hence, we start each analysis evaluation with the GQM plan. An overview of the GQM plans is given in Figure 7.1, 7.2, and 7.5. The figures use a shortened version of the GQM approach for a better overview. The detailed explanation can be found in the corresponding subsections. The figures contain for each evaluation goal a tree. The root node is the evaluation goal. Then the evaluation questions are shown. Because of space reasons, they are not formulated as a question. Below the evaluation question, the metrics answering the

questions are shown. After the introduction of the GQM plan, we describe the *Evaluation Design* and discuss the *Evaluation Results* for each evaluation goal. Each analysis evaluation concludes with a discussion about the *Threats to Validity*. We categorize each threat to validity section according to the categorization by Runeson et al. [157]. We described the four categories previously in Section 2.2.1.

After the individual analysis evaluation sections, we discuss in Section 7.4 the *Assumptions and Limitations* of our approach and summarize the evaluation results in Section 7.5.

7.1. Usage Analysis

This section describes the evaluation of the access usage analysis introduced in Section 5.1. It evaluates how well the research question **RQ1.2** is answered by the contribution C3. In addition, it indirectly evaluates the contribution C1 by using the metamodel in the analysis and therefore exploiting the pragmatics of the metamodel. The evaluation is based on our original application of the approach [214].

7.1.1. Goal, Question, Metric

The evaluation is based on the evaluation in our publication for the scenario-based access usage analysis [214], and we follow the GQM approach. This section will describe the goal, questions, and metric used in the evaluation. Figure 7.1 illustrate our GQM plan graphically. Our evaluation goal is: **G1** Validate the *accuracy* of the access usage analysis results from a security expert, software architect and domain expert view This evaluation goal describes how close our analysis results are to the real results. These real results are also often described as ground truth. Ideally, the ground truth is independent of the analysis and describes the ideal system reactions. The ground truth is, in our case, the expected results from the scenarios. In other approaches, similar properties are also called the "quality" of the results [36]. This name illustrates the importance of this goal. A high accuracy is desirable for the analysis since a low accuracy results either in overseeing confidentiality violations or blocking legitimate requests. In the first case, the access policies are too open. However, the analysis might not identify a problem. Hence,

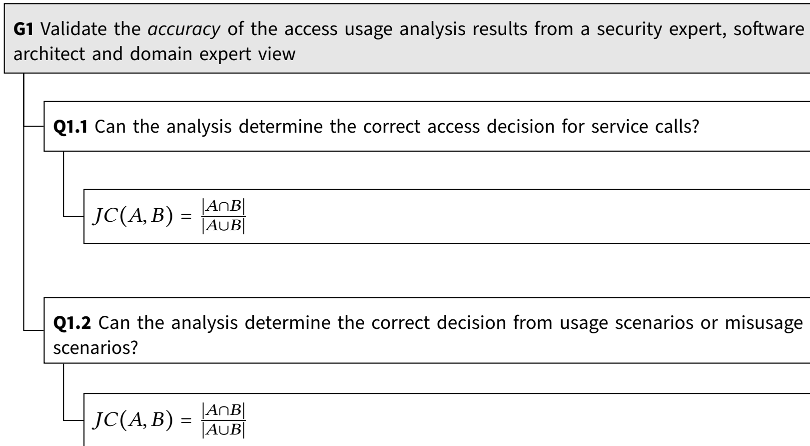


Figure 7.1.: Overview of the GQM plan for contribution C3

malicious users could get access to confidential data. This could lead to fines by data protection agencies or, for instance, in the production sector to the loss of trade secrets, which gave a competitive edge against competitors. The second case can lead to a similarly bad result. In the worst case, a user could be hindered entirely by blocking legitimate access right. For instance, in our running example, falsely denying the service technicians access to the `Terminal` can hinder them from fixing the broken machine. In this case, the production would continue to be stopped, which can be very costly. Even more, the company might decide to widen their access policies so that no legitimate access is blocked. However, this might lead to the first type of problem. In addition, a low accuracy can affect other quality attributes, such as applicability or usability since users might decide that the results are not reliable enough to be considered. Hence, it is important to investigate the accuracy of the analysis. In addition, related approaches, such as Seifermann [166], Busch [36], and Pilipchuk [143] also often consider this property.

Based on our evaluation goal **G1**, we derived the following evaluation questions:

Q1.1 *Can the analysis determine the correct access decision for service calls?*

Q1.2 *Can the analysis determine the correct decision from usage scenarios or misusage scenarios?*

The evaluation questions are helping us to narrow down what we understand under accuracy. The first question **Q1.1** concentrates on the concrete access decision for service calls. The analysis determines whether the call is permitted or denied for every service call within a scenario based on the modelled access control policy. These service calls cover not only the `EntryLevelSystemCalls` from a scenario, but they also cover delegated system calls, which are performed during an `EntryLevelSystemCall`. These system calls, in conjunction with the attributes derived from usage or misusage scenarios "simulate" the usage during runtime. In other words, they are the potentially called services by users and malicious users. Hence, these access decisions should be correct to have an overall high accuracy. A potential low accuracy in this question can indicate problems with the context derivation and access determination. Even more, it could indicate shortcomings in the modelling if, for instance, certain attributes necessary for a correct decision cannot be considered. In addition to the identification of analysis and modelling problems, a low accuracy in **Q1.1** propagates to the overall results since they build up on it.

The second evaluation question **Q1.2** builds up on the previous question and its results. In **Q1.2**, we investigate whether our analysis determines correctly that a usage or misusage scenario is marked as passed. As described in Section 5.1.2, a usage scenario is marked as passed if all contained service calls are permitted. In contrast, as described in Section 5.1.3, a misusage scenario is marked as passed if at least one contained service call is denied. It is important to consider this aspect, in addition to **Q1.1**, since the usage or misusage scenarios group together user behaviour. If this grouping is done along the use cases of a system, each usage or misusage scenario could describe a use case or misusecase of the system. This can help architects or experts to see quickly in which cases there is a problem. The service results alone cannot provide this since the service calls are not grouped. In addition, the service call itself does not store whether it is supposed to be denied, such as with a misusage scenario or whether it should be permitted. Hence, with the result alone, software architects or experts cannot determine whether a denied service call is a problem or not. A low accuracy can also be problematic since our result model is structured based on this grouping. Therefore, a wrong result might unintentionally hide potential confidentiality incidents from security experts or architects.

For answering our evaluation questions **Q1.1** and **Q1.2**, we compare our results to the ground truth. This comparison is performed by comparing the

sets of our ground truth to the result sets. For quantifying this comparison, we need a metric which can compare two sets. In this goal, we choose the Jaccard Coefficient (JC) [102] defined as:

$$JC(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

It compares the two sets A and B for equality by dividing the power of the cut of both sets divided by the power of the union of both sets. If both sets are equal, the result is 1.00. If there is no intersection, the result is 0.00. As a drawback, the JC does not consider the order of both sets. So if the order within the sets is essential, the metric cannot be used. In our case, this is not important. Our reason will be described in more detail in Section 7.1.2. Other evaluations of PCM-based analyses, such as Heinrich [70] or Monschein et al. [119], use the metrics in a similar manner.

7.1.2. Evaluation Design

We design our evaluation based on the described GQM plan for the access usage analysis. For the evaluation goal **G1** with its two evaluation questions, we need to determine the reference set or ground truth. Similar to case studies, which can provide better insights, show the applicability, and increase the comparability [49], we use scenarios to determine the comparison set. In our case, we use four of our evaluation scenarios. These evaluation scenarios are the *TravelPlanner* (Section 6.1), *ABAC-Banking* (Section 6.5), *Education* (Section 6.6), and *Maintenance* (Section 6.7). We used these scenarios because they are used in related approaches, such as Seifermann et al. [168] or Fisler et al. [57]. The other scenarios from our evaluation scenarios (c.f. Chapter 6) are not applicable because they were missing detailed behaviour descriptions necessary for the modelling and evaluation.

For each evaluation scenario, we manually created the reference output as a ground truth. We derived the manual reference output based on the source description and the modelled software architecture, together with the access control policies and the usage and misusage scenarios.

For answering **Q1.1**, we first identified all the services which are called within the usage and misusage scenarios. For each instance of the service calls, we manually determine the current context attributes which are used for the

request. Afterwards, we determine whether the call, based on the access control policy, should be permitted or not. We then store the result together with the scenario, the connector, the intended service, the origin service and the involved instantiated component. This builds a set of reference tuples for all called services. A similar set with the same tuple types can be constructed based on the analysis results. For this, we can look at the result model. It contains with the `OperationOutput` already an element which contains most of the required information. Missing are the origin service and the scenario. However, both can be reconstructed. The scenario can be reconstructed by the containment from the `ScenarioOutput`, which stores a reference to the scenario. The called service can be either reconstructed by the used connector or by using the instantiated component and reversely identifying the callee. This way, we have two sets with similar tuples. Because of our usage of additional attributes, such as components for the identification of service call instances, the order of the service call and access decision is, in our scenarios, not relevant. Therefore, we have two order-independent sets, which allows us to use the JC to compare them.

We can reuse the calculated reference set for **Q1.2** to answer the evaluation question **Q1.1**. We can group this set by the scenario. This allows us to manually decide whether a scenario is passed or not. Afterwards, we store, for each scenario, a tuple with the scenario and whether the scenario is passed or not. This builds then our reference set. The same can be done by the results of our analysis. `ScenarioOutput` contains both attributes necessary to create a similar tuple. This leads us to have two sets, a reference set and the set of our analysis. The order of the tuples within a set is irrelevant for the results since the elements are identified by the scenario and the according decision, which is contained in the tuple. Hence, we can use the JC for comparison since the sets are order independently.

For the scenario analysis, we made sure that we had at least one usage scenario and one misusage scenario based on the description of the case study. This guarantees that we have both types of results (permit and deny) in the evaluation. This is beneficial since this avoids that the analysis could return always permit, and the evaluation scenario would classify this as correct behaviour. For the same reason, we have at least one failed misusage and usage scenario in the evaluation. This guarantees that we have both types of results (passed and not passed) in the evaluation, and the analysis cannot pass the evaluation by only returning the same value.

7.1.3. Results & Discussion

The results for **G1** are shown in Table 7.1. It shows for each scenario and each evaluation question the resulting JC. Overall, we had for all results a JC from 1.00. It means that our analysis can successfully identify the expected results in all scenarios.

We will now discuss the results in more depth. The results for **Q1.1** are perfect results. We can achieve these results since the scenarios are relatively small and they do not contain many ambiguous definitions. In addition, we only consider the access result with deny or permit. This boils down to a binary decision. Therefore, the results are simplified since we do not have multiple different result types. These perfect results might indicate a high specialisation of the analysis to the investigated scenarios, similar to overfitting in machine learning. We tried to mitigate this by using different evaluation scenarios and will discuss it further in our threat to validity section (c.f. Section 7.1.4). These results mean that every access request in our evaluation scenarios is correctly decided. In other words, the request generation with deriving the context and identifying the request attributes work in these scenarios correctly. In addition, the result determination, together with the result interpretation, works correctly in these scenarios. This enables architects to analyse different service calls regarding deny and permit decisions.

The results are also perfect for **Q1.2**. We can achieve this by having again a binary decision with only *passed* and *non-passed* together with a small scenario size. Especially the small scenario size might be responsible for this. Similar to the previous results, this might raise questions regarding the generalisability. We will discuss these in our threats to validity section (c.f. Section 7.1.4). The results for **Q1.2** mean that our analysis can correctly derive, based on the access decision of service calls within a usage or misusage scenario, the overall decision for the usage and misusage scenarios. This helps the software architects or experts to decide whether the intended usage is possible or whether malicious users can perform certain activities within the system.

Considering the results of both evaluation questions **Q1.1** and **Q1.2** together, our analysis enables architects to analyse different usage and misusage scenarios regarding access control violations. This enables them to investigate alternatives like different access control policies, usage, or misusage scenarios. This is useful for investigating what-if cases. In addition, it can help to

Table 7.1.: Evaluation results regarding the JC for **Q1.1** and **Q1.2**

	Scenarios			
Evaluation Question	**TravelPlanner**	**Education**	**ABAC-Banking**	**Maintenance**
Q1.1	1.0	1.0	1.0	1.0
Q1.2	1.0	1.0	1.0	1.0

harden the system by systematically defining more restrictive policies and validating whether usage scenarios are still possible. Hence, it helps to apply the *Least Privilege* [159] principle. Even more, applying this principle can be done already during the design time since no fully implemented system is necessary. The analysis only requires service specifications, the intended usage and misusage, the access control policies for the services and a specification of the instantiated components. This is also helpful during the system evaluation, where new usage scenarios or policy changes can be evaluated. In addition, the existing misusage and usage scenario can help to not forget other aspects of policy changes like the unintentional enabling of malicious usage behaviour. In other words, the approach can help to prohibit exploiting policy changes for malicious usage.

7.1.4. Threats to Validity

As described in the introduction of Chapter 7, we discuss our threats to validity based on the guidelines for case study validity from Runeson et al. [157] because the evaluation scenario share properties with case studies (c. f. Chapter 6). Hence, we use the same guidelines as for case studies to discuss the threats to validity. The discussion is split into four categories.

Internal Validity As described, this category discusses that only the expected factors influence the results. Based on the evaluation question **Q1.1**, the result is influenced by the manual creation of the result set, which depends on the access control decision. The access control decision is influenced by the request attributes and the PDP. For the first, a wrong or missing attribute can result in a wrong access decision. In the second case, the PDP could be wrongly implemented and produce wrong access control decisions. However, this would result in a set with the wrong elements in both cases. If this is the case, our JC cannot be 1.0 since the set contains different elements than the expected set. Hence, we assume the risk to be low. In addition, for the latter case, we lowered the risk by using an external PDP, which is open source and used in different projects, such as ONAP[1]. This should reduce potential implementation problems since the PDP is well established.

[1] URL: `https://www.onap.org` (visited on 04/28/2023).

Regarding **Q1.2**, the results of **Q1.1** influence **Q1.2** since the usage and misusage scenario results build upon the access control decisions. Hence, a wrong access control decision can falsify the results for **Q1.2**. However, we assume the risk to be low since we got a JC of 1.0 for **Q1.1**. Similar to the first evaluation question, another influence can be the wrong calculation of the set based on the results, i.e., the decision when a scenario is considered passed or not. We assume the risk to be low since a false result might result in a different element in our results set. Hence, the result set and the expected would differ. In this case, we could not get a JC of 1.0 for **Q1.2**. In our case, we got a JC 1.0. Therefore, we assume the risk to be low.

A threat for both evaluation questions could be the wrongly created reference set or ground truth. Even more, most of the reference sets are manually created by ourselves. A wrongly derived result element could falsify the result and result in a higher JC, despite that the results are wrong. We tried to lower the risk by using mostly external evaluation scenarios and scenario descriptions to derive the expected set. In addition, we clearly described the used scenarios and the expected results in the scenario description. This should reduce errors during the creation of the expected sets because it reduces possible ambiguities.

Another threat in the same direction is the used models. In our evaluation, we only evaluate the results of the analysis. However, the results depend highly on the used models and, therefore, on the metamodel. If the metamodel does not cover an aspect, the analysis cannot consider this. We tried to lower the risk of overseen metamodel elements by using specialised metamodels for different aspects and modelling different external scenarios. For the first case, we use the PCM for the architectural models. The PCM is already a well-established ADL and used in different analyses, such as Reussner et al. [154], Busch [36], or Seifermann [166]. Hence, we assume the risk to be low that important aspects of component-based architectures are overseen. The access control metamodel is built up on the well-established industrial standard XACML. Therefore, we also assume the risk for overseeing attributes to be low. In addition, we reduce the risk of overseeing aspects by using evaluation scenarios from related work.

Despite the usage of external scenarios, our evaluation scenarios are still comparably small. Smaller scenarios might lead to the issue that we cannot observe all functionality or side effects. Larger scenarios might contain side effects which can not be observed in small scenarios. Hence, the evaluation

with small scenarios might not show the complete picture. However, in our case, we wanted to explicitly reuse existing evaluation scenarios from related approaches. The evaluation scenarios specify the size of our investigated scenario. In addition, the chosen scenarios cover the important functionality of the approach, such as the context derivation, transformation of the access control model, access control decisions, and misusage and scenario evaluation. Therefore, investigating larger scenarios with more architectural elements might increase the result set but would not necessarily gain more insights. Hence, we assume the risk to be low. Nevertheless, there is still a slight risk that new unknown side effects occur in larger systems.

External Validity As mentioned before, this category covers the generalisability of the results. Similar to the usage of case studies, we might increase the insights into a problem by using a scenario. Yet, a scenario is only a single instance of the global problem and only shares a subset of the characteristics. Therefore, other important aspects might not be considered within a scenario, which results in a non-representative scenario. In these cases, the results might not be transferable to other scenarios. To mitigate this risk, we choose mostly external scenarios. Most of the scenarios are based on related approaches, such as Kramer et al. [99] or Katkalov [90] for the *TravelPlanner*. Other scenarios are built on evaluation cases in related work, such as *ABAC-Banking* on Seifermann [166], or the *Education* scenario on Fisler et al. [57]. The remaining *Maintenance* scenario is built up on a scenario from industrial partners in a former industrial research project [12]. Using multiple external scenarios lowers the risk of having only a very specialised set of characteristics. In addition, the different types of scenarios (research and industrial project) reduce the risk. Nonetheless, the chosen scenarios so far only indicate the functional correctness of the analysis and not the correctness of the approach in general. In the future, it might be beneficial to investigate further different scenarios or perform an industrial case study to support further conclusions.

Regarding the transferability of the results to other analysis types, we cannot draw a general conclusion. The findings or results could be transferred to similar analyses, especially the finding of **Q1.1**. In fact, we reused parts in our contribution C4.1 since it also covers the aspect of access control for services. However, other transferability requires further investigation.

Construct Validity As described, this category discusses whether the evaluation construction allows the drawn conclusions. In our case, we answer our evaluation questions **Q1.1** and **Q1.2** by using the JC. In other words, this section discusses the appropriateness of the used metric to the evaluation goal to answer the evaluation questions. Using GQM plan can reduce the threat because by highlighting the relationship between the evaluation goals and metrics, meaningful metrics can be found. In the following, we will discuss the appropriateness of the metrics in more detail.

For the evaluation, our goal is to show the accuracy of the results. We defined accuracy as how close the results are to the reference set for a given evaluation scenario. In other words, we compare two sets, the results set to the reference set or ground truth. In summary, we need a metric which describes the relationship between two sets. Hence, we choose the JC because it exactly quantifies the difference between two sets. One threat to the appropriateness is the disregard of the order during the comparison. However, our sets are order-independent as described in Section 7.1.2. Therefore, this threat does not apply in our case. Other related model-based approaches, such as Heinrich [70] or Monschein et al. [119], use the JC similarly. In addition, we used it in Walter et al. [209] to compare reference sets with results sets. Furthermore, we use the same metrics in the original publication [214] of our contribution C3. In summary, we think the metric is appropriate for the intended goal and the risk for the construction to be low.

Reliability As described, this category describes whether other researchers can reproduce the results. We increase the reproducibility twofold by first using statistical metrics and secondly by having a complete dataset [208].

The usage of statistical metrics can avoid subjective interpretations since other researchers can observe the metrics and draw their conclusions based on them. It also helps to compare the results to other results. The dataset [208] covers multiple aspects for reproducibility. Our dataset contains all the used models, the source code and binary code, as well as the expected results. Hence, other researchers can verify the models and directly use them to verify our results. In addition, the dataset provides automatic test cases for verifying the results.

7.2. Attack Propagation

This section describes the evaluation of the attack propagation analysis introduced in Section 5.2. It evaluates how well our research question **RQ2.2** regarding the attack propagation is answered by our contribution C4.1. In addition, it evaluates the contribution C1 together with C2 by using the metamodel in the analysis and therefore exploiting the pragmatics of the metamodel. The evaluation is based on the evaluation of our original publication in Walter et al. [211].

7.2.1. Goal, Question, Metrics

Similar to the previous section, we structure our attack propagation evaluation based on the GQM approach [19]. This section will describe the goals, questions, and metrics used in the evaluation. A schematic overview of the evaluation goals is shown in Figure 7.2. Our evaluation goals are:

G2 Validate the accuracy of the attack propagation analysis from a security expert and software architect view.

G3 Validate the effort reduction for the attack propagation analysis against a manual analysis from a security expert and software architect view.

G4 Validate the scalability of the attack propagation analysis from a security expert and software architect view.

The intention for **G2** is to identify how close our estimated attack propagation is to the actual attack propagation. Similar to the accuracy (**G1**) in the evaluation of the scenario-based access analysis (cf. Section 7.1), we need to determine for the accuracy a ground-truth. In our case, this is the expected attack propagation from our evaluation scenarios. A low accuracy has two disadvantages. First, the attack propagation does not identify all affected architectural elements and secondly, the attack propagation marks not-affected elements as affected. In the first case, experts using the analysis could oversee potentially affected elements. This can either let them exposed to security vulnerabilities, which attackers could exploit in a real attack. Even more, experts would not be aware that there could be a problem and might not regularly check the elements for manipulation or disregard reports indicating security incidents. On the other hand, if the analysis is

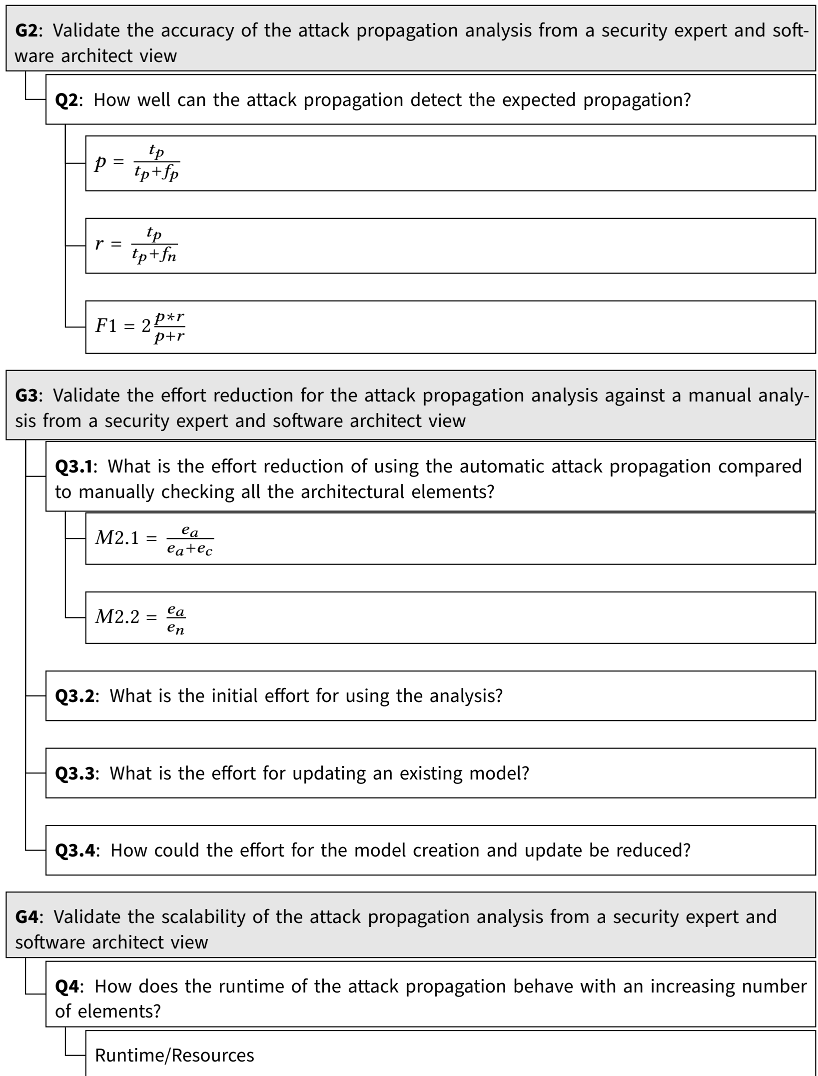


Figure 7.2.: Overview of GQM plan for contribution C4.1

used in the aftermath of an attack, overseen architectural elements could enable attackers to survive cleaning operations in the network. The second disadvantage of a low accuracy is marking non-affected elements as affected. While the security implications are not as bad as in the first case, this still has negative aspects, especially regarding the applicability and usability for security experts and architects. Many of these, also called false positives, can generate additional work for security experts since they have to manually check them for security threats in case of a breach, or they invest additional effort to secure a non-relevant element. In the worst case, software architects or security experts might dismiss legitimate results since they assume that they are false positives. Hence, a high accuracy for the attack propagation analysis is desirable. Furthermore, this property was investigated by other related approaches, such as Seifermann [166], Busch [36], and Pilipchuk [143]. However, they investigate accuracy for the prediction of other quality properties. In our case, we investigate this for the attack propagation. We also investigated the accuracy in our publication about the attack propagation analysis [211].

The evaluation question for **G2** is:

Q2 *How well can the attack propagation detect the expected propagation?*

The question narrows down our accuracy goal. The question **Q2** concentrates on the detection of affected architectural elements by an attack propagation. The analysis returns a list of affected elements for a given attacker model and initial start point. These affected elements are the components, hardware resources, network resources and services. They represent the potential compromised elements in a software architecture. Hence, as previously described, it is useful to have a high accuracy.

We answer **Q2** by comparing the results of the analysis with the expected reference result. In other words, we use the ground truth defined by our scenarios and compare it against the analysis results. During the comparison of the expected results with the analysis, we classify each architectural element from the result. Each affected result element, which is also in the expected result, is classified as a true positive t_p. Each element which is marked as affected in the result, but it is not marked in the expected set as affected, is considered a false positive f_p. Elements, which are affected in the expected result, but are missing in the result set are classified as a false negative f_n.

Based on this classification, we then calculate, precision (p), recall (r) [203] and the harmonic middle F1 of both:

$$p = \frac{t_p}{t_p + f_p} \qquad r = \frac{t_p}{t_p + f_n} \qquad F1 = 2\frac{p * r}{p + r}$$

The precision quantifies how many of our affected elements are really affected. Higher values are better here since they show that many of the found elements are really affected. Hence, it can also increase the applicability since a higher value increases the confidence of the expert that there is a security issue. The recall quantifies how many of the real affected elements can be found. Here, it is important to have high values because a low value indicates that our analysis is missing elements which an attacker can compromise. As described previously, missing these elements can have severe consequences for the security. Hence, it is important to consider the recall of our analysis. The third metric, F1, combines both values for a better comparison. Using these metrics gives a more detailed insight into how our analysis behaves in contrast to the JC from **G1**. It gives especially more insights regarding the false positives and false negatives, which are important factors in C4.1. Also, other approaches, such as Seifermann [166], use explicit metrics to give insights about the false positives and false negatives. The metrics precision and recall are also used in the foundational approach for the maintenance propagation [36, 155, 71] and other related approaches, such as Boltz et al. [32] and Walter et al. [210]. In addition, we used it in our initial attack propagation publication [211].

G3 covers the effort reduction in comparison to a manual analysis. Considering the effort for an approach is essential since a high effort for an architect might reduce the applicability and increase the costs, for instance, due to the invested time by experts. Usually, automating a previously manual process reduces the time effort. However, in some cases, the benefit comes with a higher effort in other dimensions, such as, in our case, the initial effort to model the software architecture. In this goal, we want to investigate how the effort changes by using our analysis. A similar goal is also investigated in Busch [36], where the goal is called coverage. However, they investigate the effort reduction for maintenance tasks and not attack propagations. We have also investigated a similar goal in the original attack propagation publication [211].

Based on the evaluation goal **G3**, we derived the following evaluation questions:

Q3.1 What is the effort reduction of using the automatic attack propagation compared to manually checking all the architectural elements?

Q3.2 What is the initial effort for modelling a system?

Q3.3 What is the effort for updating an existing model?

Q3.4 How could the effort for the model creation and update be reduced?

The evaluation question **Q3.1** investigates the effort in comparison to a manual analysis. In our case, the effort is the number of architectural elements a security expert has to consider for estimating an attack propagation. We assume that the security experts already have a software architecture and use the given architecture to estimate the propagation. They need to consider every element, whether it is affected by the propagation or not. Every element they do not need to manually check saves effort. Another case is the usage in the aftermath of a successful breach. There, security experts need to manually verify each architectural element for malicious changes, such as changed credentials or malware. The effort depends on the number of elements to check. Hence, if the analysis can mark elements as unaffected, these elements do not need to be checked. This saves effort and lets experts focus on the critical affected elements. We measured the effort by counting the architectural elements (`AssemblyContexs`, `LinkingResources`, `ResourceContainers`). Using counted elements enables a direct comparison without considering the experience of security experts. For the effort reduction, we use two metrics, $M2.1$ and $M2.2$, with e_a as the number of affected elements, e_c for the number of connected elements to the affected elements, and e_n as the number of all elements.

$$M2.1 = \frac{e_a}{e_a + e_c} \qquad\qquad M2.2 = \frac{e_a}{e_n}$$

The first metric describes the ratio of elements, which need to be investigated during the next propagation step. In other words, this is the effort reduction if an expert does know that no further propagation is possible for a given architecture. For $M2.1$, higher numbers are better. The second metric calculates the overall ratio between the affected elements and all architectural elements. This shows how many elements are affected. This is useful, for instance, in case of a breach, since it shows how many elements a security expert needs to investigate compared to a naive approach that investigates all elements. This metric depends on the fact that after an attack, the system needs to be cleaned of the attacker's influence. Each compromised architectural element can be a

potential starting point for another attack propagation. Therefore, security experts need to clean the architectural element. This cleaning is often done by completely resetting the device, for instance, through reinstalling and configuring the software. Hence, this is a manual effort, and it is beneficial to only clean the real affected elements. Sometimes, this cleaning also includes investigating the concrete activities of the attacker on the architectural element. This process is also often very complicated, and it is beneficial to only perform it on the affected elements. This process is even more complicated because malware can hide itself with anti-forensic tools such as the BlackPos malware[2], which attacks POS devices. For this metric, lower numbers are better.

The other evaluation questions for **G3** are slightly different from the previous ones. They are more a discussion about the involved properties and not answered by a metric. In the previous evaluation question, we neglected to consider the initial effort to create the model. In **Q3.2**, we will discuss the initial modelling effort for the usage of our analysis. Discussing the initial effort is important because creating models can be very cumbersome and time-consuming. If the initial effort is very high, it might prevent the usage of the analysis despite its benefit. Hence, we will discuss what, in our opinion, is the initial effort.

However, the initial creation effort is not the only effort for the analysis and models. Also, the effort to keep the models consistent with the real system needs to be considered. As stated in Lehman [100], software systems need to change to stay useful. Therefore, they evolve to support new functionality or use cases. Even if the system does not change, the context of the system changes. For instance, new vulnerabilities are discovered. These changes need to be reflected within the modelled software architecture. **Q3.3** investigates/discusses the effort to consider these changes.

The last evaluation question **Q3.4** discusses further potential aspects for effort reduction. Due to some changes in the metamodel or some automatic approaches, it is possible to reduce some effort for the model creation and keep it up to date.

[2] *These Guys Battled BlackPOS at a Retailer*. Feb. 4, 2014. URL: `https://krebsonsecurity.com/2014/02/these-guys-battled-blackpos-at-a-retailer/` (visited on 06/06/2023).

The last goal **G4** covers the scalability of our approach. The ongoing digitalization and trends like *Industry 4.0* or the *IoT* increase more and more the system sizes. This increase also affects the software architectures. Modern software architectures using cloud infrastructure and architectures styles like microservice can have hundreds of different services. For instance, a retailer mentioned in Newman [122, p. 6] has around 450 microservices. Depending on how abstract the software architecture is, these services would be at least 450 components. For our previous analysis, the runtime or scalability is not a very important property because it is a design time approach. Ideally, after the initial design, it only runs for evolutionary changes, such as a new usage scenario or a policy change. While these changes happen regularly, they are usually less frequent. On the other hand, for the attack propagation analysis, the scalability is more important. For instance, on average, there have been around 50 new vulnerabilities per day in 2021 [151]. To address this challenge of new vulnerabilities, it is important to run nightly or weekly security analyses, as recommended by the US American Cybersecurity and Infrastructure Security Agency (CISA) [164]. This would then provide regularly information about potential attack propagation. Of course, such a system would also require additionally the automatic updating of the models. Nevertheless, as a pre-step, it requires that our analysis can provide sufficiently fast results. In addition, a faster reaction time may increase the usability since architects or experts can quickly model alternative scenarios and investigate the output. Also, a faster analysis can help to prevent further attack propagation in case of a breach. In addition, related approaches, such as Polatidis et al. [148, 147], Ibrahim et al. [76], and Sheyner et al. [171], investigate similar properties. Hence, it is important to consider the scalability of larger software architectures. The evaluation question for **G4** is:

Q4 How does the runtime of the attack propagation behave with an increasing number of elements?

Here, we investigate the runtime of the analysis and observe how it changes with different software architecture sizes. Architecture size means the number of architectural elements, such as hardware devices or network devices. This will give us insights into how the analysis scales for large architectures. As a metric, we use the runtime in relation to the input size.

7.2.2. Evaluation Design

The evaluation design is based on the GQM plan for the attack propagation described in the previous section (Section 7.2.1). Our evaluation goals **G2** and **G3** are investigated by using the following scenarios: *TravelPlanner* (Section 6.1), *Target Breach* (Section 6.3), *Power Grid* (Section 6.2), and *Maintenance* (Section 6.7). We choose those evaluation scenarios from our described evaluation scenarios because they contain information regarding an attack propagation. In addition, the *Target Breach* and the *Power Grid* scenario are built up on real-world breaches.

For answering the evaluation question **Q2**, we need the expected set of the affected architectural elements for a given software architecture and attacker. We derived this information based on our chosen scenarios. The first step for the creation is the derivation of a software architecture. In some cases, we could reuse existing models, such as with the *TravelPlanner*. In the other cases, we manually derive the software architecture by analysing the existing artefacts. For instance, for the *Power Grid* scenario, the report from Hamilton [68] is used. Afterwards, we had for all scenarios a software architecture. We enriched these software architectures with security information, such as vulnerabilities and access control properties. The access control properties are derived from the textual description or from other source artefacts, such as existing access control policy files. The vulnerabilities are created based on the reports[3] [68, 145, 173, 170] describing the incidents or existing reports about common vulnerabilities, such as the OWASP 10 [137]. In addition, we selected for each scenario manually an attacker model. This attacker model is derived from the same source as the vulnerability model. This results in a matching attacker and vulnerability model. Based on these models, we then manually classified each architectural element. We considered during the propagation the affected components, hardware resources, network resources, services and the gained attributes for the access control. The propagation is calculated based on the attacker model and the software architecture. In some cases, like in the *Power Grid* scenario, a report [68] also states the attack propagation. In these cases, we verified the expected propagation with the described propagation. In the other cases, we could only manually verify

[3] B. Krebs. *Inside Target Corp., Days After 2013 Breach.* Sept. 21, 2015. URL: `https://krebsonsecurity.com/2015/09/inside-target-corp-days-after-2013-breach/` (visited on 09/01/2021).

the expected results. The details and characteristics for each scenario are described in Chapter 6. After the creation of the expected result, we run our analysis for the given scenario with an attacker with the same capabilities and knowledge. We then compare the analysis results with the expected result and classify the result elements. The classification of the elements is based on the previously described classification with false positives, false negatives, and true positives. We perform this classification for each evaluation scenario. We also considered different evaluation scenarios for the *TravelPlanner*. We used the additional scenario to analyse certain edge cases of the analysis, which are not included in the other scenarios. We handled these additional *TravelPlanner* subscenarios like separate scenarios. Hence, each subscenario has its own classification. Afterwards, we calculate the precision, recall and F1 measure based on the classification.

From the GQM (c.f. Section 7.2.1), we derive the evaluation question **Q3.1**. For answering the evaluation question, we utilised the results from **Q2**. First, we determined e_a by filtering the attack propagation for components, hardware resources and network resources. The size of the filtered set was then e_a. Secondly, we determined e_c. For this, we calculated the potential next propagation step. This is done by identifying the connected unaffected elements and then adding these to a connected set. Of course, in reality, this connection step would not be possible since the propagation already stopped and the attacker has not the capabilities to exploit them. However, to realise these lacking capabilities, a security expert has first to manually check the capabilities for the propagation. Hence, this would be the last step in a manual propagation, and our automatic analysis would save a security expert from doing this. Therefore, we add these elements to our connected set. Afterwards, we calculate the size of the set, which is e_c. This is done for every evaluation scenario and also for every subscenario. e_n is the size of the set of components, hardware resources and linking resources for each input scenario. After determining these values for each scenario and in the case of the *TravelPlanner* for each subscenario, we calculate the effort reduction by using our two metrics, M2.1 and M2.2, for each scenario or subscenario.

For **Q3.2**, we explain and discuss which model elements are initially involved in order to execute our analysis. The initial effort of using our analysis depends mostly on the creation of the input, as in our case, the input models. The other steps for the initial usage are setting up the tooling itself and creating an execution configuration. Both can be considered much smaller. The setup can be nearly automated because the analysis itself is fully integrated

within an *Eclipse* product. Hence, the setup only requires a compatible *Java* version and a platform where running *Eclipse* is possible. This is usually rather easily possible because *Eclipse* is a common tool used in software development. The other setup part, besides modelling, is the launch configuration creation. This consists of only creating the configuration by clicking on a button and selecting the models. Hence, we assume that the significant effort is the model creation.

For the evaluation question **Q3.3**, we discuss which elements of our metamodel are system-specific or independent. As described in the previous paragraph, we assume that the effort for generating the models is the most significant effort. Based on this assumption, the discussion about system-specific or independent model elements is beneficial because system independent elements can be used in other systems. Hence, we have the effort to create the element only once. We will describe the effort on different evaluation scenarios and discuss what effort they may create.

The evaluation question **Q3.4** is answered by discussing potential approaches to reduce the initial effort and the effort in evolution scenarios. Here, we again focus on the effort of creating the model. We especially focus on the modelling effort for the users of our approach.

We perform a scalability analysis for answering **Q4**. During the scalability analysis, we measure the runtime of the attack propagation analysis with different input sizes. We scale the input sizes based on a very simplified architectural model. It is based on the *TravelPlanner* model. However, the concrete model is not relevant for this goal because, for the measurement, we slightly modify it to support the different input sizes. This modification is the essential characteristic of this evaluation question. The modification can also easily be done to other models so that they share the same characteristic. Hence, we assume that the *TravelPlanner* is, in this case, a good representative scenario.

The first step for the scalability analysis is to determine the elements for scaling. Here, it is important to identify the relevant factors of the runtime and scale them appropriately. Based on our Algorithm 3 and 4, we can identify I) the loop (l. 4 in), II) selection of elements, III) and the compromisation of elements as an influencing factor. The other factors are, in our opinion, not relevant, because they are only executed once.

The first factor, the loop, influences the result because it is repeatable executed. Meaning all the operations within the loop are repeatable executed. These operations are the factors II and III. Therefore, our goal is to increase the number of loop iterations because this can increase the runtime of the analysis. The number of loop iterations depends on the number of propagation steps for an analysis execution. Hence, if we increase the propagation steps, we increase the number of loop iterations. The runtime of a loop iteration is dependent on the runtime of the propagation steps. The propagation steps are similar as discussed in Section 5.2.4. Each propagation step is structured similarly. It consists of two parts. First, the selection of the element (factor II) and second, the compromisation of the selected element (factor III). The selection (factor II) is the identification of the neighbouring elements based on the already affected element. This selection should be runtime-wise very similar for all propagation steps because it only follows the different connection relationships. In addition, a propagation step is always a propagation between two architectural element types. Therefore, if we want to increase our factor II, we need to increase the number of architectural elements which can be considered during the selection. However, because we assume the behaviour for the selection is similar, we can increase the number of any architectural element. Therefore to increase the problem size, we can scale along any architectural element.

The second part (factor III) is the decision on whether the selected architectural elements are compromisable. This includes for our analysis, also marking them as affected if they are compromisable. In our analysis, we have two compromisation types: the compromisation by vulnerabilities and the compromisation by credentials. Both types can influence the runtime. The runtime of the compromisation through vulnerabilities depends on how often the check for compromisation is performed and how many vulnerabilities one selected architectural element has. We increase the number of checks by increasing the number of architectural elements which need to be checked. We can achieve this behaviour similarly to the factor II by scaling the number of architectural elements. In our case, the type is irrelevant for checking the vulnerability because the structure for the checks is the same for each architectural element type. The other aspect affecting the runtime of the vulnerability compromisation is the number of vulnerabilities for one element. The reason is that the analysis implementations get a list of vulnerabilities for each selected architectural element. The analysis then iteratively checks whether any of the vulnerabilities can compromise the selected architectural

element. Therefore, if no vulnerability can be compromised, it needs to iterate over the complete list of vulnerabilities. However, the vulnerabilities for one architectural element are usually not significant. Hence, we assume that this should not be a problem and do not consider it in our factors.

The same also holds for the credentials. Usually, the access decision depends on the number of attributes relevant to the decision. Nevertheless, here the number is usually also a lower number because of the complexity of the specification. In addition, the runtime for access decision is based on the implementation of the used PDP. Hence, increasing the number of attributes would result more in a scalability analysis for the PDP than our developed approach. Therefore, we did not consider this.

Based on our findings, we found out that each of our factors I – III depends on the number of architectural elements. In addition, based on our assumption that the finding of elements should behave similarly, we identified that the concrete type of the architectural elements is not relevant. Therefore, we can choose to scale along any architectural element type. In our case, we choose to scale `ResourceContainers` because they can easily be scaled. For achieving the propagation, the `ResourceContainers` need to be vulnerable, so the propagation steps in the loop can be executed multiple times. The results for access control should be similar and only differ based on factors for each propagation step. We add the new `ResourceContainers` by adding for each new one a unique `LinkingResource`. We have then, for each new `Resource-Container`, a unique network. The `LinkingResources` are not vulnerable in our case. Afterwards, we connect each new `LinkingResource` with a `Re-sourceContainer` from another network. This builds up to a chained network of `ResourceContainers` similar to Figure 7.3. The Figure 7.3 illustrates the created chain. The `LinkingResources` are indicated by the stereotype `Network` and the `ResourceContainers` by the stereotype `Device`. The connections are the dashed lines. Each of the `ResourceContainers` is vulnerable to the same vulnerability. Hence, it creates for the attack propagation analysis something like a worst-case analysis, if the first `ResourceContainer` is vulnerable to an attack from the attacker. It forces the analysis to perform at least n propagation steps with n as the number of `ResourceContainers` in the chain. In Figure 7.3, this is illustrated by the red arcs below the devices. In the propagation step p_{m-1} where m is the current propagation, the analysis would propagate to r_j where j is the index to identify the resource. In the next step, it would identify the connected elements. In our case, this would be n_i and r_{j+1}. The other elements like n_{n+1} or r_{j+2} are not reachable since they are

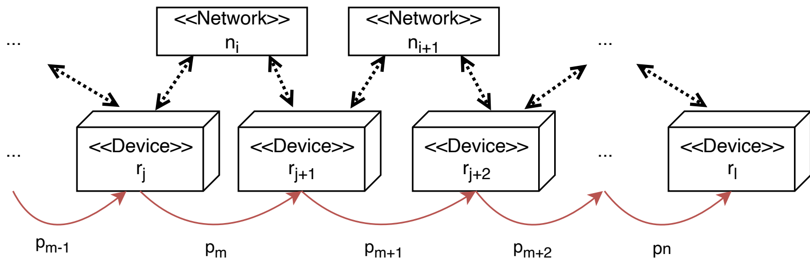


Figure 7.3.: Chained `ResourceContainers` schematics for the scalability evaluation

only transitive connected by the connected elements, and the propagation step only uses directly connected elements. From the directly connected elements, only r_{j+1} is vulnerable. Therefore, the analysis compromises in the propagation step p_m only r_{j+1}. The process is repeated, and in p_{m+1}, the now directly connected r_{j+2} is compromised. This is repeated until the last element in the chain r_l is reached. We scaled the `ResourceContainers` along the power of 10 from 10^1 elements till 10^5 elements because we assume this is a reasonable upper limit for manually modelled software architectures. We measured the time from after the loading and creation of the model till the execution of the attack propagation was finished. This measurement does not include the writing of the output on the disk. Therefore, the measurement contains only the analysis runtime with no input and output handling. The attacker model contained the capability to exploit the vulnerability of the `ResourceContainers`. In addition, we manually verified in a test run that the attack propagation marks the chained `ResourceContainers` as affected. To ensure accuracy and eliminate outliers, we repeated each measurement five times and calculated the average. In our sample data, the averaging of five proved to be enough to avoid outliers. Additionally, we performed one warm-up analysis before executing the scalability analysis. The analysis was executed on a VM with 20 AMD Opteron Processor 8435 with 62.5 GB RAM, and the operating system was Debian 11 with the OpenJDK 17.

7.2.3. Results & Discussion of Accuracy

We investigated 18 scenarios for the attack propagation analysis (one scenario for each real-world breach, 15 *TravelPlanner* subscenarios, and the *Mainte-*

nance scenario as the running example). The results for **G2** are shown in Table 7.2. It shows for each evaluation scenario and evaluation subscenario the answer to each corresponding evaluation question. The first column *EQ* lists the evaluation questions. The second column contains the metrics (*M*) to answer it. Afterwards, the columns for the scenario and subscenarios are listed. The column *T* is the *Target* scenario, the *P* is the *Power Grid* scenario, all the columns starting with *TP* are the *TravelPlanner* subscenarios, and *MT* is the *Maintenance* scenario. Overall the results for **G2** are very good.

In detail, for **Q2**, we have a precision (p), recall (r) and F1 from 1.00 for all scenarios. These are perfect results for our scenarios. We can achieve these perfect results since in the real-world based scenarios (*Target*, *Power Grid*), the system was highly compromised. In addition, the subscenarios in the *TravelPlanner* scenario are very small due to their focus on edge cases. Both aspects reduce the complexity of the expected outcome and enable us to achieve these. In other scenarios, the result can differ. However, these simplified scenarios enable us to get a better understanding of functional properties, such as wrong propagation rules. In our threat to validity section (c.f. Section 7.2.6), we discuss the implication of the small evaluation scenarios in more detail. The evaluation results indicate that the analysis implementation and propagation rules are, at least for the investigated scenarios, the required ones. They also indicate that we can use the analysis in similar scenarios and expect good results. Nevertheless, the very good results raise questions regarding too high specialisation of the analysis for the investigated scenarios and how generalisable the results are. We discuss this in our threats to validity section (c.f. Section 7.2.6).

Overall, based on our results, we can say that our attack propagation analysis can help identify potentially affected architectural elements by attackers. The results of **Q2** indicate that the affected architectural elements are accurate for similar scenarios like in our evaluation. Based on our scenario choices, we cover different domains, such as enterprise backend applications or IoT environments. Hence, we assume that the results are transferable to other scenarios. In addition, the security experts also get first insight into the potentially affected data.

Based on these results, security experts or software architects can harden the software architecture by introducing appropriate mitigation concepts to prevent the attack propagation. By hardening different parts of the software

architecture and preventing the propagation of attackers, the overall security of the system can be increased.

7.2.4. Results & Discussion of Effort Reduction

The second evaluation goal for the attack propagation analysis is the effort reduction (**G3**). It is investigated by our evaluation questions **Q3.1**,**Q3.2**, **Q3.3**, and **Q3.4**.

We start by discussing the results for **Q3.1**. Table 7.2 shows the results for **Q3.1** for each investigated scenario, where *T* is the *Target* scenario, *P* is the *Power Grid*, the *TPs* are the subscenarios from the *TravelPlanner*, and *MT* is the *Maintenance* scenario. For *M*2.1, the results in the *Target* and *Power Grid* scenarios are lower than in the *TravelPlanner* subscenarios. This observation also reflects the expected outcome because of the number of affected elements. In the first two scenarios, nearly all the architectural elements are affected. There the number of unaffected elements is reduced, which leads to more elements for the security expert to check manually. Therefore, the effort reduction is reduced in comparison to a fully manual analysis. This manual analysis is necessary to further clean a system after a compromisation (c.f. Section 7.2.1 – **Q3.1**). The reduction can be seen even more if we investigate the resulting attack propagation in detail. In both scenarios, the unaffected elements are mostly external elements, such as outside network elements. Hence, the effort reduction might be slightly less than described in real systems. This effort reduction depends on the fact that the security expert needs to clean the compromised elements from malicious users manually. Therefore, if the attack propagation is smaller, fewer elements are affected, and fewer elements need to be manually cleaned. This illustrates the *TravelPlanner* subscenarios where the effort reduction is higher. This is based on the fact that in these scenarios, the attack propagation does only affect a part of the system. Hence, the automatic analysis reduces more elements for a manual inspection and can therefore save more manual effort in these scenarios. Because in *TP1* no propagation happens, we cannot calculate a value for it. Nevertheless, the result indicates, based on the *TravelPlanner* subscenarios, that for smaller breaches which do not affect the complete system, our analysis can save between 44% and 88%. For *M*2.2, the results look similar. The scenarios with a smaller propagation indicate better results. These are shown by the low number in most cases. It indicates that in certain

Table 7.2.: Evaluation results for the attack propagation analysis regarding accuracy and effort reduction

		Scenarios																	
EQ	**M**	**T**	**P**	**TP1**	**TP2**	**TP3**	**TP4**	**TP5**	**TP6**	**TP7**	**TP8**	**TP9**	**TP10**	**TP11**	**TP12**	**TP13**	**TP14**	**TP15**	**MT**
Q2	p	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	r	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	F1	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Q3.1	*M2.1*	0.13	0.28	—	0.86	0.63	0.78	0.78	0.78	0.67	0.70	0.71	0.67	0.44	0.60	0.80	0.83	0.88	0.88
	M2.2	0.87	0.68	0.00	0.10	0.30	0.20	0.20	0.20	0.30	0.30	0.20	0.30	0.50	0.40	0.20	0.10	0.10	0.13

scenarios, the security experts can save significantly their effort by using the attack propagation. In some subscenarios like, for instance, TP2 or TP14, the value is 0.10 very good. This results in an effort reduction from about 90% to a fully manual analysis since only 10% of the elements need to be manually analysed. As previously described, the manual analysis covers the clean-up operation of compromised architectural elements (c.f. Section 7.2.2). Overall the results indicate a potential effort reduction in comparison to a fully manual analysis. However, this is not always the case for systems with a higher percentage of affected elements. Nevertheless, even in these scenarios, the attack propagation analysis can help the security experts by getting first insights about the propagation or verifying their manual attack propagation. Hence, in cases of breaches, our analysis can help to secure the system or, during the design time, show the experts some insights regarding possible attack propagation without manually estimating the attack propagation.

Here, we discuss the initial effort necessary to model a system for using our analysis and thereby try to answer the evaluation question **Q3.2**. The initial effort can also be described as activities necessary to perform for executing our analysis. To answer this evaluation question, we focus on the creation of a fully specified model. In the evaluation question **Q3.4**, we will also discuss how different abstraction levels can help to reduce the effort. Table 7.3 shows the activities based on the involved model elements. The first column shows the PCM viewpoint if the involved models are PCM elements. In the case of our models, it shows the involved metamodels. The second column has the model elements, and the next column marks whether the model element is affected. For this evaluation question, the column *Initial* is the last relevant one. A line marked with x states states that the model element needs to be created.

Overall all elements are marked as required. The involved activities are, first, the creation of the *Service types*. In PCM, this is the definition of interfaces and services. Then the creation of model element *Component types*, which is the creation of components in the repository, such as `BasicComponent` and the selection of provided and required interfaces. The model element *Components* describe the creation of instantiated components in the system and their connection to other components. The actual services implementation is the model element *Services*. This covers the SEFF specification in PCM. Our analysis also uses the information from the *Deployment*. Hence, the model elements for the *Devices*, *Network* (the connection between network nodes and devices), and the `Allocation` need to be created. So far, these are “only” PCM

Table 7.3.: Activities performed during the usage of the attack propagation analysis

Category	**Model Element**	**Initial**	**S**	**E1**	**E2**	**E3**
Structural	Service types	x		x		
	Component types	x		x		
	Components	x	x	x		
Behavioural	Services	x		x		
Deployment	Devices	x				
	Network	x				
	Allocations	x	x	x		
Propagation	CWE/CVE IDs	x		(x)	x	
	Vulnerabilities	x	(x)	(x)	x	
	Roles	x	x	(x)		
	Communication	x	x	(x)		
	Attacks	x			(x)	
	Attackers	x	(x)	(x)	(x)	(x)
Access Control	Attributes	x	(x)			x
	Attribute Providers	x	(x)	(x)		
	Access Control Policies	x	x	x		x

model elements. The next model elements are the new elements introduced by our analysis. The first is the creation of model element *CWE/CVE IDs*. These are the IDs later used in the *Vulnerabilities*. Here, the vulnerabilities must be modelled. For this, the security experts need to analyse the existing architectural elements, identify the vulnerability, and assign them to the architectural element. The next two activities add additional metadata to the architecture by identifying potential roles between components, such as clients and servers. In addition, they need to specify which components can communicate globally. The model element *Attacks* is the identification and creation of existing attacks. The last activity in the propagation section is the modelling of the model element `Attackers`. Here, security experts need to model the attackers. The next group is the *Access Control* activities. Consisting of the creation of the model elements *Attributes*, assigning the model element *Attribute Providers* and the definition of the model element

Access Control Policies. Overall to answer our evaluation question **Q3.2**, the analysis needs, in the worst case, 16 different model elements. Hence, the effort for creating the system is higher than for only software architecture models with PCM, which requires only 7 elements. However, in the other evaluation questions, we describe how the initial effort can be reused or even lowered.

While the initial effort can be very cumbersome with the creation of the different model elements, many of the model elements can be reused, for instance, in evolution scenarios. With the evaluation question **Q3.3**, we discuss what effort is needed in evaluation scenarios and what elements can be reused. We split the discussion into two parts. First, we discuss system-specific and independent model elements. Afterwards, we discuss different evolution scenarios and what model changes they would require.

The fourth column illustrates system-specific model elements (*S*). We mark system-specific elements with x. Some elements have aspects which are system-specific and some elements are system independent. These elements are marked with (x). The elements without a marking are system independent. System independent elements are model elements which can be reused in other systems. For instance, the component types are system independent, meaning, that a modelled component can be reused in a different system requiring the same functionality. For the elements of the original PCM categories *Structural*, *Behavioural*, and *Deployment*, we reused the existing classification described in Reussner et al. [154, p.44]. For the others, we manually derived them. The *CWE/CVE IDs* are in general system independent because CWEs are describing by nature system independently weaknesses and CVEs describe a concrete vulnerability of an element. While the element can be system-dependent, the ID for the vulnerability itself is not. The *Vulnerabilities* are system-specific and system independent. Similar to the CVE, the vulnerability itself is system independent. However, the attributes regarding the gained credentials are system-specific. In addition, in some cases, other attributes of the vulnerability can be system-specific. For instance, the attack vector can vary in some cases on the concrete configuration of a software product. However, these are very rare edge cases where the general vulnerability classification differs from the concrete one. The *Role* assignment is again system-specific, as well as the *Communication*. The attacks are like the CWE system independent because the attack contains only exploitable IDs. The *Attackers* itself have both parts again. The system-specific part is the starting point of the attacker and the knowledge about attributes. The other parts of

the capabilities are, in general, transferable. The *Attributes* are, in general, system independent. For instance, the role model of a system can be transferred between different systems. Also, other aspects, like the state attribute, can be used in a different system. However, there are exceptions. For instance, the issuer flag can have system-specific attributes. The same is true for the *Attribute Providers*. In case they use system independent elements, they can be system independent. Otherwise, they are system-specific. The *Access Control Policies* are generally system-specific. They protect system-specific architectural elements or the rules are specific to the use case. Nevertheless, there might be sometime cases, such as a very identical system with the same organisation hierarchy, where parts can be reused. In summary, our approach has many system independent elements, which can be reused in other models. Therefore, for many elements, the effort is only an initial effort, and the effort can be reused in evolution steps.

To further discuss the evolution effort, we introduce three evolution scenarios based on typical scenarios during the evolution. We consider these scenarios because they represent regular activities which should be performed for active systems. However, we do not intend to claim completeness on the scenarios. There can exist other scenarios.

E1 Updating the software architecture, by adding a new component and services to the system.

E2 Adding a new vulnerability to the software architecture

E3 Adding a new user role to the access control policies

The involved model elements in the evolution scenarios are shown in the three last columns from Table 7.3. The involved model elements are marked with x. In some cases, the involvement is dependent on the concrete situation. In these cases, we mark them with (x).

The first evolution scenario **E1** represents the typical scenario that new functionality should be added to the system. For instance, a web server could be added to our running example. This would require adding the new component and the provided server, resulting in effort to update these models. In addition, the new component must be deployed somewhere. In our case, we could deploy it on existing hardware. Therefore, the *Allocations* need to be updated. Depending on whether the new components have vulnerabilities, also the *CWE/CVE IDs* and *Vulnerabilities* need to be updated. The same holds for *Roles* and the *Communication*. Both need to be updated if the new

component adds either a new role or cannot communicate with everyone. Depending on whether the new component can also provide certain attributes, the `Attribute Providers` needs to be updated. In addition, the *Access Control Policies* need to be updated to grant access to the new component.

The second evolution scenario **E2** represents the case that a new vulnerability is found for an existing architectural element. This is a common use case, especially for longer-running systems, where new vulnerabilities are found over time for the architectural elements. In this scenario, in the best case, we only need to update the *Vulnerabilities* and *CVE IDs*. In some cases, we might also add a new attack and add this to the attackers. However, this is optional. For instance, if the new vulnerability belongs to a CWE class for which an attack already exists, it is unnecessary to define a new attack. Nevertheless, there also might be a scenario where security experts explicitly want to create a new attack for this vulnerability, for instance, for high-profile vulnerabilities.

The last evolution scenario **E3** represents a scenario where a new user role should be added to an existing access control model. In this scenario, we assume that the role is not stored on a component, and our architecture abstraction does not include identity providers. In this case, we need to update the *Attributes* by adding the new role. In addition, the *Access Control Policies* need to be updated so that they use the new role. Optionally, the *Attackers* can be updated. This is necessary if the new role changes the threat model. For instance, the new role is a newly contracted company and not very much trusted yet. Here, the security expert can create a new attack with the knowledge of the company and analysis the impact.

Overall for **Q3.3**, we have seen that not all elements need to be updated in the different evaluation scenarios. Many elements can be reused or stay the same. Even during larger changes, which affect multiple model elements like in **E1**, many of these elements are optional and depend on the concrete scenario. It can probably be assumed that in these cases, often not all optional model elements need to be updated.

Our last effort evaluation question **Q3.4** covers the discussion about potential effort reductions to our approach. We identified three aspects that can help reduce the effort during our work. As for the previous effort discussion, we do not claim completeness, but the aspects are rather three example cases of how the effort can be additionally reduced.

1) Different abstraction levels
2) Automatic software architecture recovery and update
3) Automatic vulnerability updates

The first part is about the different abstraction levels. In **Q3.2**, we discussed which steps are necessary for creating an initial model. During the discussion, we explicitly stated that these steps are necessary for fully modelled software architectures. However, in some cases, we do not need a fully modelled software architecture or even cannot provide one, for instance, during the early design time. Also, the abstraction can be helpful. One obvious answer to how abstraction can save the effort is that a higher abstraction layer combines multiple smaller components into one component and, therefore, reduces the number of components needed to be modelled. However, the same can also be true for a partial model where we abstract from certain aspects. For instance, our attack propagation does not necessarily need the specified services. Therefore, software architects can leave them out and still use our analysis. The same is true for hardware resources and deployments. Theoretically, the propagation also works only on components or only on connected resource containers (similar to our scalability evaluation). This is especially useful in different design phases, where certain information is not yet available. For instance, if the hardware is unknown, it is already possible to propagate on the component architecture. This way, software architects can potentially save some modelling effort for the software architecture and security experts need to identify fewer vulnerabilities. It also stretches the modelling effort over a longer time. However, like always with a higher abstraction, there might be the case that through the abstraction, some security violations are not identified.

The second part to reduce the effort is automatic architecture recovery tools. The model creation is a big effort, especially for old and larger legacy systems. Often these systems also do not have accurate models, or there exists no model or documentation at all. There exist approaches which can help in these cases. These approaches recover the software architecture by using the source code, investigating log files, or other build artefacts. Typical approaches for PCM are SoMoX [25], Monschein et al. [119] or Kirschner [93]. In general, they provide not the exact software architecture but a representative software architecture. Also, they currently provide only the software parts. They cannot provide information about the hardware resources. Also, some of these approaches cannot yet partially update the model, which is useful

in evolution scenarios. Nevertheless, Monschein et al. [119] and Heinrich [70] can provide these features. By integrating their approaches, it might be possible to reduce the effort further because then the architecture model would be automatically updated.

The third part of the potential effort reduction is the automatic derivation of vulnerabilities. In Section 4.3.5 and in Kirschner et al. [94], we already presented an initial approach for automatically creating the vulnerability model. While it cannot derive some system-specific information for the vulnerabilities, such as the gained attributes, it can already automatically derive most attributes of our vulnerability model. Therefore, it can help to significantly reduce the effort for the initial creation of the vulnerability model. Currently, it does not support updating the vulnerability model. However, it might be possible to implement this feature in combination with approaches, such as iObserve [70]. Therefore, this might reduce the effort in evolution scenarios.

7.2.5. Results & Discussion of Scalability

In this section, we present and discuss our findings about the evaluation goal **G4** regarding the scalability. We discuss it by answering our evaluation question **Q4** with a scalability experiment based on the scaling of `ResourceContainers` and measuring the runtime.

The measured runtime for the scalability is illustrated in Figure 7.4. The vertical axis contains the runtime in ms. It shows a logarithmic scale from 10^2 till 10^9 ms. The horizontal axis contains the number of `ResourceContainers`. It is a logarithmic scale from 10^1 to 10^5. The line with the circles shows the average runtime of the attack propagation analysis. Each circle represent a scaling step. For instance, we measured the average runtime at 10^1 and 10^2. The line between two datapoints (circles) is interpolated from the datapoints. For 10 elements, the analysis needs around 555 ms. The runtime then slowly increases to 2,918 ms for 100 elements and 52,905 ms for 1000 elements, which is roughly a bit less than one minute (52 seconds). Afterwards, the runtime drastically increases for 10,000 elements to around 3,864,488 ms, which is a bit more like one hour. It then further increases for 100,000 elements to around 442,632,110 ms, which is around five days. As expected, the number of vulnerable `ResourceContainers` affects the runtime of the attack propagation analysis. We also assume that the propagation

for other architectural elements will produce similar results because the propagation rules for the `ResourceContainers` are structurally similar to the propagation rules of other architectural elements.

The analysis runtime is a significant challenge, particularly for architectures with a large number of elements. The execution time required by our analysis does not support prompt responses to newly discovered vulnerabilities, nor does it currently allow for daily analysis runs in large systems. Instead, it suggests that a weekly analysis run may be more practical. However, daily analyses are feasible for smaller systems containing 10,000 elements or fewer. For example, the shop system with 450 microservices mentioned in Newman [122, p. 6] can be analysed within a reasonable timeframe. The runtime should be within minutes if the model is not highly detailed such as only one component for each microservice. One possible solution to improve scalability is to create a more abstract architectural model that reduces the number of architectural elements. This approach is also applicable in domains, such as IoT, where similar sensors can be grouped together as a single sensor. Although the model size is reduced, the analysis results remain similar since the sensors share the same characteristics. Moreover, the results of [56] suggest that the average time frame for exploiting a vulnerability is six days. Thus, our analysis could potentially identify a potential attack propagation before an attacker can exploit them. However, the trend towards faster exploitation of vulnerabilities highlights the need to improve scalability in the future.

As for potential improvements in runtime, several options are available. First, the used CPU architecture (AMD Opteron Processor 8435) is quite old because it was released in 2009. A newer CPU architecture might produce slightly better results. In fact, initial testing on newer CPUs demonstrated some improvement of around two-thirds, particularly for smaller systems with 10,000 elements or fewer. Second, parallelizing the execution of the propagation rules could reduce the runtime by making better use of available hardware resources. Currently, the application is mostly single-threaded, which limits the processing capacity. Third, the data structure for extending the software architecture and modelling it is not ideally suited to this type of analysis. The analysis requires frequent searches across different modelling domains, which can lead to slower performance due to the increased runtime by searching in lists. Using more caches or more efficient data structures for searching could address this issue. Finally, improving the handling of propagation triggers could be another area for potential improvement. Currently, after a change is

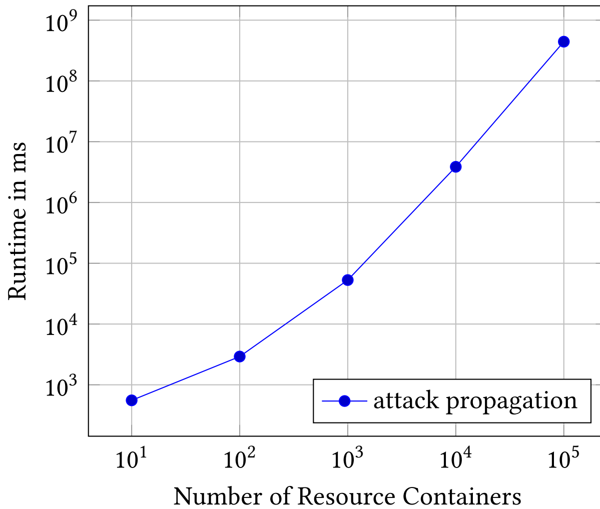


Figure 7.4.: Scalability results (**G4**) for increasing number of resource containers

made to an element, all the propagation rules are retriggered, which can be inefficient. A more effective solution would involve the fine-grained selection of relevant propagation rules and only retriggering those, thus minimizing unnecessary computations.

In summary, the runtime is for larger systems could be better. However, the runtime still enables a weekly analysis. In addition, there is potential for performance improvements for larger systems by refactoring the analysis for more efficient algorithms and data structures or using higher abstraction. In addition, for smaller architecture, the runtime is already acceptable and might improve further on current hardware.

7.2.6. Threats to Validity

Similar to the previous analysis, we structure the threats to validity after the guidelines for case study validity from Runeson et al. [157]. We transferred the guidelines to our mostly scenario-based evaluation because of the similarity to case study-based research. The four used categories are the *Internal Validity*, *External Validity*, *Construct Validity*, and *Reliability*.

Internal Validity This validity category is about the influencing factors for the results. For the evaluation question **Q2**, the results are influenced by the classification of the architectural elements as affected or not. The classification is based on the attack propagation rules, which mark an architectural element as affected or not. A possible threat could be that the propagation rules mark the wrong element as affected or not affected. In this case, the wrong element could be affected. This would result in a wrongly classified element. However, depending on the other propagation rules, this might not be uncovered only by the classification of affected and not. In some cases, other propagation rules affect the element, thereby fixing the previous propagation mistake. Therefore, the element is affected by the wrong element. However, these results can be identified by manually identifying the propagation source and vulnerability, which we did after the evaluation. During this additional manual analysis, we identified no such cases. In the other case, in which no other propagation rule "fixes" the other rule, the result would be marked as wrongly classified. In this case, our metrics for the scenario could not be 1.0. Hence, we assume this risk to be low.

The expected classification in affected elements and not affected elements could be false because we manually created them. To lower the risk, we used mostly external scenarios and created the expected results based on existing literature [170], reports [68, 173, 145], and security advisors, such as OWASP 10 [137] or the NVD [131]. In addition, based on the fact that our metamodel uses similar properties to existing vulnerability classifications, such as CVE, the assignment of the correct properties and the impact is also easier because they can mostly be directly extracted. However, this can also be a drawback if the used source wrongly classified the results. An example of this wrongful classification is the vulnerability in our running example [129]. Nevertheless, in general, we assume that the classification by experts for vulnerability classification is correct. Hence, we consider this threat to be low.

Another point affecting the internal validity based on the results and scenarios is the size of the used models. The scenarios themselves are quite small. Therefore, it might be that not all aspects of the analysis are evaluated. We mitigated the risk by introducing the specific subscenario for the *TravelPlanner* case. These cover missing aspects from the other scenarios. Overall, our used scenarios include the important aspects of the attack propagation analysis, such as gaining new credentials, exploiting vulnerabilities based on CWEs/CVEs (authenticated and not authenticated), propagation from

different compromised architectural elements, such as `AssemblyContext` or `ResourceContainers`. Therefore, adding more architectural values might not result in more insights regarding the basic functionality. Nevertheless, in larger case studies, there might be unknown side effects. However, based on our very good results for the smaller systems, we think that even if these unknown effects may lower the accuracy, it still should be good results.

Another risk is the used metamodels and propagation rules. The evaluation is based purely on the analysis results and does not separately evaluate the used metamodels and propagation rules. If the metamodel neglects a relevant property for the propagation, the propagation rules cannot use them. In addition, we might not have considered every possible propagation as a propagation rule. We reduced the risk by the metamodel twofold. First, we used the PCM as our architectural metamodel, which is a well-established ADL used for many different quality properties, including security properties like confidentiality [166, 143]. For the second part, our access control, vulnerability, attack, and attacker metamodel is built on different well-established external standards and classifications. This might reduce the risk of overseeing some aspects. Nevertheless, this comes with the drawbacks of these classifications, and this must be considered. However, similar properties to our used ones can be found in other external classifications. Therefore, we assume the risk to be low. In addition, during our evaluation, we identified no missing aspects for the scenarios.

For the effort reduction goal **G3**, our threats are aligned with the discussion about effort and its measurement. In **Q3.1**, we decided to measure the effort solely based on counting model elements. Other aspects, like the creation of the models, are not considered. We discussed this aspect in **Q3.2**. We discuss similarly the questions **Q3.3** and **Q3.4**. However, in all three cases, this is a more a qualitative discussion.

For the evaluation question **Q4**, an internal threat is that the increased measured runtime does not depend on the increased `ResourceContainers` but is affected by other attributes. The scaling factor is limited to two architectural elements to reduce the threat. We only introduce additional `ResourceContainers` and `LinkingResources`. In addition, we manually verified that the newly added `ResourceContainers` are vulnerable and affected during the attack propagation and the `LinkingResources` are unaffected. The additionally added `LinkingResource` can slightly affect the propagation runtime since

they are also checked for vulnerabilities, but their checking is similarly implemented to the ResourceContainers. In addition, a scaled real system might also have additional LinkingRessources. Also, the external insights we got through profiling the application support the aspect that an increased number of ResourceContainers affect the runtime. Besides internal factors like the discussed model input, other external factors, such as the system usage or other executed processes, can affect the scalability results. We tried to reduce these external factors by using a dedicated separated Virtual Machine (VM) only for the scalability analysis. This VM was assigned dedicated processing and memory resources to reduce the effect of other VMs running on the device. In addition, each scaling step was repeated five times, and the average was calculated from the measured data. Therefore, we assume the risk of interference to be low.

External Validity As described, this validity discusses how generalisable the results are. As mentioned in the access usage scenario evaluation, we might increase the insight into a problem by using scenarios for the evaluation. As a drawback, the scenario might be very specific and does not share all the required properties which a problem in general has. Therefore, the chosen scenarios might not be representative for the overall problem. We tried to mitigate the risk by investigating different external scenarios. Two scenarios are based on real-world breaches, one scenario is based on a common research system, and the last scenario is based on a scenario described by industrial partners in a research project. The different scenarios also cover slightly different application domains, such as Industry 4.0 or Enterprise Business Systems. Therefore, we assume the risk of too specific scenarios to be low.

Nevertheless, the investigated scenarios show more the functional accuracy than the general accuracy in real-world systems. Especially aspects like the potential overestimation due to the inference of compromised data or the handling of AssemblyContexts deployed on compromised ResourceContainers is not thoroughly investigated. Here, the main problem is to find real-world scenarios containing detailed enough information to observe this effect. In addition, in the investigated scenario, nearly all elements are affected. Therefore, the effect could not be observed. The same also holds for the effort reduction goal, which is specific to the investigated scenarios.

For the scalability goal **G4**, a threat is the generalisation from the ResourceContainers to the general scalability behaviour of the attack propagation

analysis. Internally, the `ResourceContainer` propagation is structured similarly to the other propagations rules. Therefore, the results should be transferable to the other propagation rules. Their runtime might be slightly different because they differ in small details. However, the overall timeframe should be comparable.

Construct Validity This validity discusses whether the investigated property is appropriate for the intended goal. In our case, the investigated properties are the used metrics. We used a GQM plan to reduce this threat. The GQM plan highlights the relationship between the evaluation goals and metrics. This clear structure can help to find meaningful metrics for the evaluation goals. In the following, we will discuss the appropriateness of the metrics for each evaluation goal and question.

Our first evaluation **G2** is investigated by the research question **Q2**. To answer the evaluation question **Q2**, we used the metrics precision, recall and F1. These metrics help to identify the impact of false positives, true positives and false negatives. Therefore, they give more insights regarding how accurate the results are. In addition, the metrics are used to evaluate the foundational maintenance propagation [156, 155, 71, 36]. In addition, other approaches in the field of model-driven security analysis, such as Seifermann et al. [168] or Boltz et al. [32], use identical or similar metrics. Therefore, we assume the risk to be low.

The evaluation question **Q3.1** belongs to our evaluation goal **G3**. To answer the evaluation question **Q3.1**, we use two counting metrics, $M2.1$ and $M2.2$. Both metrics are a ratio between counted model elements. The first metric calculates the effort reduction in case a security expert knows where the propagation ends. Hence, it calculates the ratio of the additional elements which need to be investigated. In contrast, the last metric illustrates the overall effort reduction if not all elements need to be checked. It is based on the evaluation of the foundational propagation approach [155], where a similar evaluation question is answered. There, they investigated an additional ratio which contained the false positives. However, in our case, this is not necessary since we had in our evaluation no false positives, which would result in the same values as we have calculated. Therefore, we assume the metric is appropriate because it describes the effort reduction overall and is already used in foundational approaches. In addition, we used both metrics in Walter et al. [211].

Regarding the scalability, we answer the evaluation question **Q4** by using the runtime in relation to the scaled `ResourceContainers`. In other words, we put the runtime in relationship to a scaled input size. Similar metrics are used in other model-driven approaches, such as Heinrich [70], for observing the scalability. Also, in related approaches, such as Polatidis et al. [148], the runtime is measured for different input scales. Therefore, we assume that the chosen metric is appropriate for the evaluation goal.

We answer **Q3.2**, **Q3.3**, and **Q3.4** by a discussion about the modelling effort. This is not ideal, and we cannot provide quantitive results. However, we choose the discussion to provide more insights about the modelling effort than not considering it.

Reliability This category discusses whether other researchers can verify and reproduce the results. Most of our evaluation questions are answered by using metrics. This usage can increase the reproducibility by avoiding subjective interpretation. It is also useful in the comparison with other approaches or extended approaches. Another aspect for lowering the threat is to use a structured evaluation plan as we did with the GQM. This helps other researchers understand the relationship between the goals and metrics more easily.

For answering **Q3.2**, **Q3.3**, and **Q3.4**, we choose a discussion about the effort necessary to create the model. We broke down the model creation into simple update or creation tasks which are associated with the involved model elements. By using these simpler tasks, we tried to lower the risk of our subjective interpretation and tried to increase the reliability.

In addition, we prepared a dataset [208] containing all the used models, source code and binary code to execute the analysis. With the dataset, other researchers can verify the model and the results. In addition, the expected results for the evaluation question **Q2** are encoded in Java rules, so others can more easily verify the results. In addition, we provide an automated process for answering the evaluation question **Q4**. This enables other researchers more easily to reproduce the results. The reproduction is also possible on different hardware as the used hardware in our case. This does not affect the evaluation goals **G2** and **G3**. For the scalability evaluation, this should only affect the overall values but not change the drawn conclusion. While the used evaluation system had more hardware resources than regular systems,

the approach does not exploit them. Therefore, any regular computer system should be sufficient.

7.3. Targeted Attack Graph Analysis

In this section, we discuss the evaluation of the *Targeted Attack Graph Analysis* (c.f. Section 5.3). During the evaluation, we discuss how well our research question **RQ2.2** is answered. In the evaluation, we cover the contribution C4.2 and the metamodel contributions C1 and C2. As in the previous evaluation, we do not directly evaluate the used metamodels but only evaluate them based on their pragmatics through evaluating the analysis. The evaluation is based on the evaluation of our original publication in Walter et al. [212].

7.3.1. Goal, Question, Metrics

Like in the previous sections, we use the GQM approach for the evaluation of the targeted attack graph analysis. In this section, we present the used goals, questions, and metrics. The GQM plan is also graphically represented in Figure 7.5. We start by explaining our evaluation goals:

G5 Validate the accuracy of the targeted attack path analysis from a security expert and software architect view.

G6 Validate the effort reduction of the targeted attack path analysis regarding manual analysis from a security expert and software architect view.

G7 Validate the scalability of the targeted attack path analysis from a security expert and software architect view.

The evaluation goal **G5** investigates the accuracy of predicting attack paths to a targeted element. A similar property is investigated in both previous evaluations (c.f. Section 7.1 and Section 7.2). Like for the **G1** and **G2**, we need to determine a ground truth for the evaluation. In our case, this is possible attack paths between different architectural elements. An analysis with a low accuracy comes with the same drawbacks as described for **G2**. As described, a low accuracy can lead to that security experts are not aware of existing attack paths. Therefore, allowing a malicious attacker to exploit

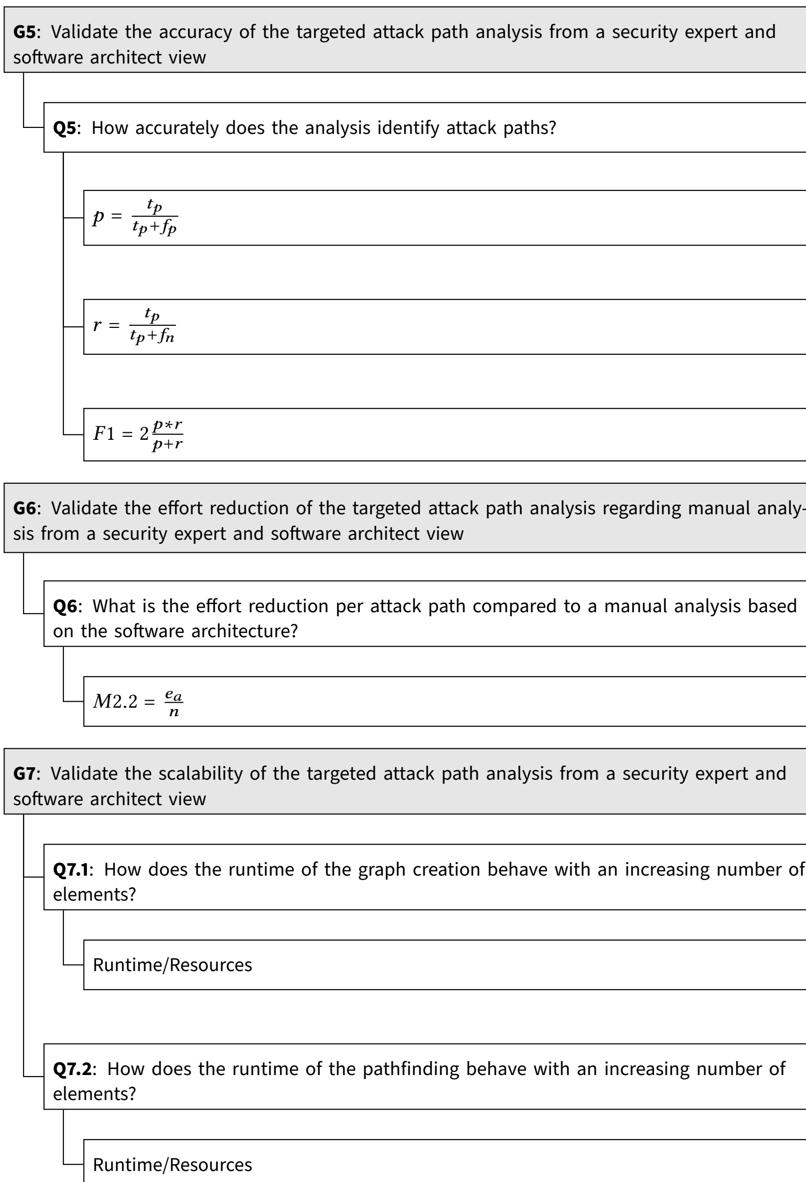


Figure 7.5.: Overview of GQM plan for contribution C4.2

them. In addition, it might be that security experts waste important resources for protecting non-affected architectural elements, which leads to additional costs or that protection resources might not be available for relevant affected architectural elements. Also, a high rate of false positives might negatively affect the security experts using the analysis. They might consider the results irrelevant and categorise them as false even when they are correct. Therefore, a high accuracy is desirable for the targeted attack path analysis. Other related approaches, such as Seifermann [166], Busch [36], and Pilipchuk [143], also investigate accuracy. However, they investigate it for other quality predictions and do not investigate the accuracy of attack path estimations.

The evaluation question investigates the accuracy of the analysis results. The evaluation question is: **Q5** *How accurately does the analysis identify attack paths?* The question focuses on the detection of attack paths to estimate the accuracy. We decided to use this question since our analysis returns a list of possible attack paths to the targeted element. The attack path contains the involved architectural elements. Here, the architectural elements are `AssemblyContext`, `ResourceContainer`, and `LinkingResource`. Therefore, these are the directly used results software architects, together with security experts, can use to increase the security of the system.

For answering the evaluation question **Q5**, we classify the analysis results, which are the list of the found attack path. A valid attack path is a true positive (t_p), which means the analysis correctly detected an attack path between a possible start point and the targeted element. In addition, the attack path is valid, meaning it contains a list of connected elements from the start point to the targeted element, which can be compromised under the filter criteria. False positives (f_p) are attack paths found by the analysis, but there exists no real attack path. False negatives (f_n) are not found or not valid attack paths by the analysis. Based on this classification of the found attack paths, we calculate our metrics, which are precision (p), recall (r) [203] and the harmonic middle F1:

$$p = \frac{t_p}{t_p + f_p} \qquad r = \frac{t_p}{t_p + f_n} \qquad F1 = 2\frac{p * r}{p + r}$$

The first metric, the precision, quantifies how many of the found attack paths are considered by the manual analysis as valid attack paths. Here, a higher value is better because it indicates a higher percentage of correctly found

attack paths. A high precision can increase the trust of security experts in the results of the analysis. Therefore, it might also increase the applicability. The second metric, recall, shows the ratio of found attack paths to non-found attack paths or invalid attack paths together with the found attack paths. Here, a higher value indicates fewer missing attack paths or invalid ones. Missing attack paths can lead to security incidents because security experts are not aware of security threats and cannot mitigate them. Therefore, a higher recall is better. The last metric, F1, combines both values to compare them. Here, also higher values are better. Overall, the chosen metrics give insights into the accuracy by considering also the false positives and false negatives. Also, other related confidentiality approaches, such as Seifermann [166], use metrics to gain insight into false positives and false negatives. Our used metrics are also very commonly considered in the evaluation of model-based quality analyses, such as Busch [36], Heinrich et al. [71], and Boltz et al. [32]. Also, we used the metrics in various confidentiality-related publications, such as Walter et al. [211, 210].

Our second goal **G6** investigates the effort reduction by using our automated analysis in contrast to a fully manual analysis. Investigating the effort is important because a high effort may prohibit the application by software architects or security experts. Therefore, ideally, the analysis should reduce more effort than it generates. Also, the related approach Busch [36] investigates a similar goal with their coverage. Also, our related attack propagation analysis (c.f. Section 5.2 and Walter et al. [211]) investigate a similar goal.

Our evaluation question for narrowing down the goal **G6** to a specific effort is: **Q6** *What is the effort reduction per attack path compared to a manual analysis based on the software architecture?* With the evaluation question, we want to investigate potential effort reduction by using our analysis compared to a complete manual analysis. Similar to the evaluation question **Q3.1** from the attack propagation analysis, we assume that software architects or security experts use in the manual analysis as well as in the automatic analysis software architecture models and security models like in our analysis. Therefore, we do not consider the initial effort to create the model or the evolution effort to update the model. However, we discuss these properties for the attack propagation in the evaluation questions **Q3.2**, **Q3.3**, **Q3.4**. We determine the effort by counting different model elements and calculating the ratio. We reuse the metric *M2.2* from the attack propagation. e_a is the number of

affected architectural elements within an attack path, and e_n is the number of all architectural elements.

$$M2.2 = \frac{e_a}{e_n}$$

The metric calculates the fraction of affected architectural elements within a path in contrast to the overall architectural elements. In a manual analysis, a security expert would need in the worst case analyse all the involved architectural elements. In contrast with our analysis, they only need to consider the elements marked by the path. Here, a lower value is better.

For the effort goal, we also need to consider similar evaluation questions regarding the modelling effort as in **Q3.2**, **Q3.3**, and **Q3.4**. In our case, we do not investigate these questions in this section because this analysis uses for the vulnerability the identical metamodel to our attack propagation analysis. The discussion about the effort reduction would be nearly identical because nearly the same steps are involved. The main difference is that we do not need to model the attacks in this analysis. In addition, the attacker model is slightly different because we do not model concrete capabilities. Nevertheless, the discussion is transferable to this analysis. Therefore, we refer to Section 7.2.4 for an in-depth discussion about the modelling effort.

The last evaluation goal **G5** covers the scalability of our approach. It should investigate the runtime behaviour of our approach for larger systems. Due to trends like IoT or Industry 4.0, the systems contain more and more entities. These entities are also reflected as devices and components in the software architecture. The reaction time of the analysis does not need to be within a range of seconds. However, a fast reaction time is beneficial to continually search for new attack paths and react to newly found zero-day exploits. For instance, a typical use case might be the daily analysis for newly found attack paths similar to nightly builds running integration tests in the software development process. For instance, the US American Cybersecurity and Infrastructure Security Agency (CISA) recommends running nightly security analysis [164]. In addition, related approaches, such as Polatidis et al. [148, 147], Ibrahim et al. [76], and Sheyner et al. [171], investigate similar properties. Therefore, it is important to consider the runtime of the analysis.

We use two evaluation questions to narrow down the evaluation goal. Our evaluation questions are:

Q7.1 *How does the runtime of the graph creation behave with an increasing number of elements?*

Q7.2 *How does the runtime of the pathfinding behave with an increasing number of elements?*

We separate the evaluation questions into two questions along the two parts of our analysis, the attack graph creation with **Q7.1** and the attack pathfinding with **Q7.2**. By separately investigating the different parts, we get a better insight into problematic parts. As the metric, we choose for both questions the runtime in relation to the input size. Therefore, the metric is the same as in **Q4**, which investigates the scalability of the attack propagation analysis. The metric is also similar to other related approaches, such as Heinrich [70] or Polatidis et al. [148].

7.3.2. Evaluation Design

We created the evaluation design based on the GQM plan for the targeted attack path analysis, introduced in the previous section Section 7.3.2. We use the evaluation scenarios *TravelPlanner* (Section 6.1), *Cloud Infrastructure* (Section 6.4), *Target* (Section 6.3), *Power Grid* (Section 6.2), and *Maintenance* (Section 6.7) for investigating our evaluation goals **G5** and **G6**. We chose those evaluation scenarios from our described evaluation scenarios because they contain information regarding attack paths. The evaluation goal **G7** is investigated on a simplified architecture model.

To answer the evaluation question **Q5**, we need to know whether an attack path exists between two architectural elements and what valid attack paths are. In our evaluation, we start by deriving the attack paths. We used the same models for the evaluation scenarios as in the attack propagation evaluation (c.f. Section 7.2). In addition, we investigated the *Cloud Infrastructure* scenario, which stems from a related approach. For each scenario, we first manually identified potential start elements. These start elements are the architectural elements from the type `AssemblyContext`, `ResourceContainer`, and `LinkingResource`. The start points are further restricted based on the chosen potential `StartFilters`. After finding all starting points from the architectural elements, we manually determined for each start element whether there exists at least one attack path between the start element and the targeted element. The attack paths can use vulnerability or potentially

found credentials as long they are not filtered as described in the scenario description. Each scenario had at least a `CredentialFilter` excluding all credentials. This results in attack paths, which either need to use vulnerabilities to compromise other architectural elements or search for credentials on other vulnerable elements and then use these found credentials. This decision leads to more complicated attack paths. Otherwise, the analysis could just assume it gains the highest credentials, such as root or admin or use them to directly compromise elements. Afterwards, we know for which architectural element an attack path exists to the targeted element. We use this information to derive the true positives, false positives and false negatives for each scenario. If we manually find an attack path, we mark the resulting attack path from the analysis as a true positive. If our analysis does not find a path, but we manually find an attack path, we count this as a false negative. If our analysis finds an attack path, but there is no manually found attack path, we count it as a false positive.

However, we cannot yet calculate our metrics based on these values since the positive set might still contain false negatives. We need to identify whether the found attack path is valid or not. This cannot be done by comparing the manually found attack path with the automatically found attack path because they can be different paths. There can be different attack paths between the start element and the targeted element. In our manual analysis, we do not necessarily need to find the same path as the automatic analysis. For instance, we have already seen two possible paths in Figure 5.15. These paths can be restricted by filters, but this is not true for all paths. Therefore, we need either to determine during the manual analysis all possible attack paths and then compare whether one of these matches the analysis results or manually verify whether there exists an attack path and then check the validity of the found analysis attack path. The first solution requires a very high effort or might be even impossible because it requires all possible paths, and with loops, the number can be endless. Hence, we choose the second option. This gives us the knowledge that at least the manually found attack path exists. The analysis can then return the same or any other attack path. In the case it is another path, we manually verify the attack path by following it from the start element to the targeted element. For each element (excluding the start element), we checked whether it could be compromised from its predecessor. This check also included the consideration of potential filters. If all the elements could be compromised, we consider it a valid attack path. If the attack path is not valid, we remove it from the true positives and count it

as the false positives. Otherwise, we leave it in the true positive set. After doing this for all attack paths within a scenario, we have the number of false negatives, false positives, and true positives. We can then calculate our metrics precision, recall, and F1 based on these values.

We utilize the existing results from **Q5** to investigate the evaluation question **Q6**. We use the found valid attack paths and determined for each path e_a. e_a is the number of affected architectural elements within an attack path. We can determine e_n by determining the number of all architectural elements within a scenario. Based on these values, we can calculate the $M2.2$ for each attack path within a scenario.

We investigate the evaluation goal **G7** and its evaluation questions **Q7.1**, and **Q7.2** by performing a scalability analysis. During the scalability analysis, we measure the runtime of different analysis parts for different input sizes. The input model is a very simplified architectural model.

As a first step, we have to identify the influencing factors for answering **Q7.1** and **Q7.2**. For **Q7.1**, we analyse what are the key factors that influence the runtime of the graph creation Algorithm 6. We identified the four loops (l. 4, 6, 13, 19) as the main factors in our case. The other factors influence the runtime only as a constant factor. Two loops (l. 4, l. 6) are propagating over the architectural elements and the two other over the number of access control policies for one element (l. 13) and the number of vulnerabilities for one element (l. 19). Based on these loops, the first two loops are basically the number of connected architectural elements. Therefore, if we increase the number of connected architectural elements, we increase the loop iterations. This allows us to reuse the same method for scaling as for **Q4** and scale along one architectural element type. Scaling along the architectural types is also useful because these are the main structural elements within a software architecture. Hence, if the software architecture grows, the number of architectural elements also increases. Because our algorithm handles all the architectural element types similarly, which type we scale is irrelevant. The runtime for different architectural elements should be similar and may only differ in a constant factor. Therefore, we choose to scale along chained vulnerable `ResourceContainers`, which are connected by non-vulnerable `LinkingResources`. This is the same behaviour as in the previously described Figure 7.3. In this case, the start element is the first `ResourceContainer`, and the targeted element is the last `ResourceContainer`. Hence, the analysis needs to iterate over the complete chain to determine the attack path. Also, the created attack graph needs to

contain the complete chain. Therefore, this creates some sort of worst-case analysis because the complete chain needs to be considered in both cases. The two other loops can be neglected because we assume that the number of loop iterations is usually low. The number of loop iterations is not affected by larger software architectures. These loops depend not on the overall number of policies or vulnerabilities. While the overall number of vulnerabilities and access control policies might increase in larger software architectures, these loop iterations depend on the vulnerabilities and access control policies for one element. Typically, a larger software architecture does not necessarily mean that the involved architectural elements have more vulnerabilities or access control policies. Hence, we assume that for these loops, the number of iterations is still small, and we assume that they have no significant effect on larger architectures.

During the investigation of **Q7.1**, we measure the runtime from starting the graph creation till the attack graph is returned. The measurement does not include the loading of the architectural models. We scaled the number of `ResourceContainers` by the power of 10 from 10^1 `ResourceContainers` till 10^5 `ResourceContainers`.

For investigating **Q7.2**, we measure the runtime after the attack graph is created till an attack path is returned. Here, we also investigated which factors influence the scalability. Our Algorithm 7 only operates on a graph, with edges and architectural elements as nodes. Here, the important aspects are the number of start points, the number of distinct nodes which are connected by edges and the path length. In our case, we choose to neglect the start filter and only investigate scaling the other properties. We explain our reasoning regarding the start filter after discussing the importance of the other attributes. The other properties increase the runtime for finding a path because they increase the problem size. The different architectural element types are not relevant anymore, and they are abstracted as nodes. We can choose any architectural type. Therefore, if we want to identify the worst-case scenario, we need to have an attack graph, which forces the path finding to iterate over all elements. We enforce this behaviour by defining a `StartFilter` for the first `ResourceContainer` in the chain and selecting the last `ResourceContainer` in the chain as the targeted element. This forces the path finding to iterate over the complete `ResourceContainer` chain and creates a worst-case runtime. In addition, it gives us control over the attack path length, and we can investigate different attack path lengths. Otherwise, the length of the attack paths may vary on the selected start element. The start filter is

also closer to the intended usage for larger architectures. We assume that software architects are more interested in attack paths from certain elements in the software architecture to a target. Similarly, like for instance, in our running example, where we had the scenario where we were only interested in an attack path from the externally accessible component to the critical component. Even more, our investigation gives insights into an upper bound for one attack path. Finding additional attack paths should take equal or less runtime per each additional path finding.

For both evaluation questions, we repeated each measurement five times and calculated the average to reduce outliers. Additionally, we performed one warm-up run beforehand. We performed the analysis on a dedicated VM. The VM was assigned 21 AMD Opteron 8435 cores together with 62.5 GB RAM. The operating system of the VM was Debian 11, and we used the OpenJDK 17 as the runtime environment. For the analysis execution, we increased the Java stack size to 1 GB and the heap size to 30 GB.

7.3.3. Results & Discussion of Accuracy

We investigated five different scenarios containing overall 52 possible attack paths. The results for the accuracy **G5** are shown in Table 7.4. The first column shows the scenario, the second column the precision (p), the third one the recall (r) and the fourth one the F1. Overall, we got good results, although unlike for the attack propagation analysis, they are not perfect results.

In detail, we have a precision of 1.00 for every scenario, which is a perfect result. Achieving these results is possible because our evaluation scenario and attack paths are small. We will further discuss the implication of the small scenarios in the threats to validity section (c.f. Section 7.3.6). In addition, we focused on a restricted model with no dependencies to unknown behaviour, which simplifies the results. These results mean that every attack path found by the analysis was a real attack. In other words, every attack path resulting from our analysis in these scenarios is valid. Regarding the recall, we get for the *Target*, *Cloud Infrastructure* and *TravelPlanner* scenarios a recall from 1.00, resulting in an F1 score of 1.00 for these scenarios. This means that our analysis discovered all the required attack paths in these scenarios. In other words, the analysis found an attack path to the targeted element for

each potential start point, like the manual analysis. These are, again, perfect results.

However, our analysis cannot find attack paths for all required ones in the two remaining scenarios. In the *Power Grid* scenario, the analysis misses two possible attack paths, and in the *Maintenance* scenario, it misses one path. This leads to a recall of 0.88 and an F1 score of 0.93 for the *Power Grid* scenario. For the *Maintenance* scenario, the recall is 0.86, and the F1 score is 0.92. We also investigated the reason for the missed attack paths. In both cases, the missing attack paths can be traced back to the implementation of the path finding. For performance reasons, we need to restrict the path finding implementation to simple paths. This is based on the used graph framework and its path finding algorithms. Using a simple path forbids loops in the attack path. Therefore, attack paths which require a loop, for instance, to gain certain credentials are excluded. For our missing cases, this is exactly the case. For instance, in the case of the maintenance scenario, an attack path from the `TerminalServer` to the targeted `ProductStorage` requires one self-loop to get the necessary credentials. Otherwise, the attacker lacks the necessary credentials to compromise the other architectural elements. Losing this requirement in our current framework would result in a strong increase in the runtime, which renders the approach for even small scenarios nearly useless.

Based on the results for the different scenarios, we can say, that our analysis can identify attack paths in certain scenarios. The precision indicates, that the found attack paths are very likely to be real possible attacks. In our scenario, every found path was a real attack path. Even more, the recall suggests that we can identify most of the real attack paths. The analysis can miss some paths depending on the scenario, but most attack paths are identified. Overall, the results indicate that the analysis can help software architects or security experts in hardening the system by providing possible attack paths. They harden the system by mitigating possible attack paths and therefore, increasing the overall security of the system.

7.3.4. Results & Discussion of Effort Reduction

The results for the evaluation question **Q6** is shown in Table 7.4. For simplicity reasons, we only illustrate the minimum and maximum value of our metric for each scenario. Each scenario has values according to the number of attack

Table 7.4.: Evaluation results for the targeted attack graph analysis regarding accuracy and effort reduction

Scenario	p	r	$F1$	$M2.2_{min}$	$M2.2_{max}$
Target	1.00	1.00	1.00	0.13	0.27
Power Grid	1.00	0.88	0.93	0.11	0.26
Cloud Infrastructure	1.00	1.00	1.00	0.06	0.10
TravelPlanner	1.00	1.00	1.00	0.20	0.20
Maintenance	1.00	0.86	0.92	0.25	0.50

paths. The full table can be found in our dataset [208] or in the appendix (c.f. Table A.1). The minimum value is in the second last column, and the maximum value is in the last column in Table 7.4. The values vary overall between 0.06 for the *Cloud Infrastructure* and 0.50 for the *Maintenance* scenario. Depending on the scenario, they vary around two times between the minimum and maximum value, with the exemption of the *TravelPlanner*. The exemption for the *TravelPlanner* can be traced back to, that it contains only one attack path. Hence, there can be no different values. The variation in the other scenarios depends on the length of the different attack paths. Architectural elements which are more remote to the targeted elements, in the sense that more architectural elements are necessary to reach the target, produce a higher value. Despite the different attack path sizes, the values are still close together. This closeness is based on the used attack path finding algorithm. Our algorithm is a modified shortest-path algorithm. Therefore, the algorithm tries to produce short paths and usually only includes the minimal necessary attack step. Therefore, it does not include many unnecessary architectural elements. The result is that even the maximum paths are short and directly to the targeted element. Hence, the values are still close since no “detours” are performed.

Overall the results indicate an effort reduction since in most of our scenarios only around a third or fewer elements need to be manually verified in comparison to a complete manual analysis.

7.3.5. Results & Discussion of Scalability

In this section, we present and discuss the findings about our evaluation goal **G7** by answering the evaluation questions **Q7.1** and **Q7.2**. We performed a scalability analysis by increasing `ResourceContainers` and measuring the runtime.

The runtime measurement, together with the increased `ResourceContainers`, is illustrated in Figure 7.6. The vertical axis shows the runtime in a logarithmic scale from 10^1 ms to 10^7 ms. The horizontal axis contains the number of `ResourceContainers` in a logarithmic scale from 10 to 10^5 `ResourceContainers`. The average runtime for the *graph creation* is shown by the line with the dots. The line with the squares shows the average runtime for the *path finding*. Each dot or square is an average measurement point. In other words, these are the model sizes where we measured the runtime. The lines between the measurement points are interpolated based on the measurement points. For both evaluation questions, the runtime is very close together. The evaluation question **Q7.1** is answered by the `graph creation`. For 10 elements, the graph creation needs around 26 ms. It then slowly increases to 106 ms for 100 elements and to 597 ms for $1,000$ elements. Afterwards, the runtime growth increases more. For $10,000$ elements, it needs already $24,681$ ms. The last measurement was for $100,000$ elements, which took around $5,764,543$ ms. This is around 1.5 hours for the creation of the attack graph with $100,000$ `ResourceContainers`.

The evaluation question **Q7.2** shows similar behaviour. The answer is shown by the *path finding* curve. The runtime starts a bit higher with around 42 ms for 10 elements. It then slowly increases for 100 elements with 75 ms to $1,000$ elements with 693 ms runtime. It then rapidly increases the runtime. For $10,000$ elements, it takes $48,512$ ms and for $100,000$ elements, $5,653,641$ ms. So, the finding of an attack path in a system with $100,000$ `ResourceContainers` also takes around 1.5 hours, like the graph creation.

Overall, this summarizes to a runtime of about 3 hours for finding attack paths. Our findings are based on the scaling of `ResourceContainers`. As expected, the scaling shows that the `ResourceContainers` influence the runtime. Our assumption is that the other architectural elements behave similarly. The observed runtime behaviour does not enable an immediate reaction to newly found vulnerabilities in larger systems with 100,000 or more affected elements. Nevertheless, it still indicates that, for instance, the analysis can be executed

as a daily nighttime job, where the newly found and disclosed vulnerabilities are evaluated. This is then similar to nightly build jobs, which usually execute integration tests. In our case, this is then a security analysis. In addition, it might be possible to reduce the model sizes. For instance, in IoT environments, not every sensor or actor instance needs to be modelled in the software architecture. As long as they share the same vulnerabilities and tasks, it might be sufficient to group elements of the same functionality into one element. A simple example could be in our running example, the replication of the machine. There could be multiple instances of the same machine, but the security results would be in the single instance and multiple instances similar. For smaller systems with $10,000$ or fewer elements, the runtime is good enough to provide nearly immediate feedback with a combined runtime of 1.3 minutes or less. For instance, depending on the abstraction level, the shop system mentioned by Newman [122, p. 6] with around 450 microservices can be seen as ideal size for fast analysis results.

If we analyse the runtime in more detail, we also see potential improvements like for the previous analysis with **Q4**. The first one might be the age of the used CPUs. Despite that during the graph creation 21 cores are used, the cores themselves are old. The CPU architecture was initially released in 2009. Therefore, newer CPUs might provide better results. In addition, there might be some improvements with caches or more efficient algorithms and data structures as also described for **Q4**. Another aspect is the single-thread process for the path finding. Currently, we parallelised along the number of attack paths. In other words, each path finding algorithm is executed as a separate thread. Therefore, the finding of multiple paths should scale along the number of CPU cores. Here, an alternative solution can be that the path finding itself uses a multi-thread approach to faster identify an attack path. However, there might be a problem with the credential filter because this induces a required execution order.

7.3.6. Threats to Validity

Similar to the evaluation of the other analyses, we present the threats to validity in accordance with the case study validity guidelines established by Runeson et al. [157]. Given the similarities between our mostly scenario-based evaluation and case study research, we have adopted these guidelines.

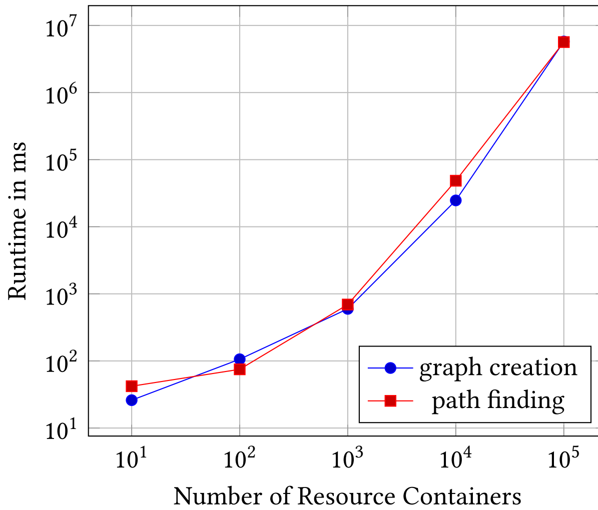


Figure 7.6.: Scalability results (**G7**) for increasing number of resource containers

Our threats to validity discussion consider four categories, namely: Internal Validity, External Validity, Construct Validity, and Reliability.

Internal Validity As described, this validity discusses the influencing factors for the result. Similar to the previous analysis, the result is directly affected by the classification necessary for answering **Q5**. Here, among others, two threats can arise. First, the manual identification of possible attack paths and second, the determination of a valid attack path. In the first threat, we could have made some errors in the determination of attack paths. This leads to either missing attack paths or wrong attack paths. Missing attack paths are especially relevant in scenarios where the number of attack paths is smaller than the possible start locations, for instance, for the *TravelPlanner*. In these scenarios, we might have overseen attack paths. We consider this threat low since our automatic analysis did not find additional attack paths to our manual analysis. Furthermore, the automatic analysis misses some attack paths. This can be a first indicator that our manual analysis provided more in-depth results. Secondly, we repeated the manual determination for the missing attack paths. In this case, we tried to mitigate possible overseen paths in the first manual run. The second threat is based on the validity of the attack

paths. We could have wrongly classified the attack paths. This could be either that we classify an invalid path as valid or classify a valid path as invalid. We consider both cases to be low. For the first case, we compared the result beside the manual step-by-step verification to the sources and descriptions of the scenarios to determine whether the attack paths are plausible. Also, the manual analysis is very straightforward since it only requires manually verifying that the next element is reachable and whether the attack is possible with the chosen filters. Here, a benefit is the small size of the scenarios since it makes the manual analysis easier, therefore, helping not to oversee elements. We consider the second case, where we classify a valid path as invalid, also as low since we have a precision of 1.0. Therefore, all found attack paths are marked as valid, and we did not falsely mark a scenario as false.

Another threat is the model size and attack path length. While the small size helps, in the classification above, it might prevent the observation of side effects, like in the case with simple paths for finding attack paths. Therefore, there might not all aspects of the analysis be investigated. While we consider this a threat, we lowered it by choosing evaluation scenarios that already contain most of the important aspects of the analysis, such as the exploiting of vulnerabilities, using access control properties and gaining of credentials. Therefore, adding more architectural elements might not bring more insights. However, in other scenarios, the results might differ. Nevertheless, we assume that based on the high accuracy we achieved in our investigated scenarios, the accuracy in other but similar scenario types is comparable. Similar to the discussion in our other attack analyses, also the inputs could be wrong, such as in the case of the vulnerability of our running example. However, here we also assume that, in general, the input classification is correct.

In addition, a threat is the sole focus on the analysis during the evaluation. Similar to the other evaluation, we do not evaluate the used metamodel directly. As described, that might lead to the fact that we neglect relevant properties for finding attack paths. However, by reusing the ADL PCM, we lowered the risk for the software architecture. For the other metamodels, we lowered the risk by reusing most of the parts in the other analyses, which consider similar properties. Therefore, we assume the risk to be low because missing very relevant properties might also have an effect on the evaluation of the other analyses.

One effect on the goal **G5** for the accuracy is the chosen evaluation question **Q5**. The evaluation question focuses on the attack paths. However, the attack

paths are affected by the previously determined attack graph. Therefore, the accuracy for the attack paths depends also on the accuracy of the attack graph. By not directly investigating the attack graph, we might oversee aspects influencing the accuracy. Nevertheless, we choose only to investigate the paths because with the paths, we indirectly also evaluate the created graph. Despite our focus on the paths, we analysed the generated attack graph during the evaluation to find the source of missed attack paths. We looked at the generated attack graph and whether this would allow the missed attack paths. In these cases, we found no issues with the generated graph. Therefore, we assume the risk to be low.

For answering our effort evaluation question **Q6**, we did not consider the initial effort to create the models. Therefore, the results are not complete. However, automatic approaches, such as our automatic extraction approach [94], could help to reduce the effort. We further discuss some additional properties in the effort evaluation of the attack propagation analysis (c.f. Section 7.2.4).

A possible threat to the evaluation goal **G7** regarding the scalability is that the observed runtime increase is not based on the scaled `ResourceContainers`. We reduce the threat by limiting the scaling to vulnerable `ResourceContainers` and the connecting non-vulnerable `LinkingResources`. This way, only those two elements change regarding the input model between the different measurements. To further narrow down the effect, we observed the found attack paths and verified that they contained the newly added `ResourceContainers`. Nevertheless, the runtime is slightly affected by the additionally added `LinkingResources`. This effect cannot be completely removed because the analysis needs connecting elements. Also, considering other architectural elements would always result in adding an additional connecting element. However, we assume the influence factor to be low since the `LinkingResources` do not contain vulnerabilities. Besides the internal factors of the input models for the analysis, also external factors can affect the scalability analysis. In our case, this can be the system usage, other executed process or the number of available CPU cores. In our case, we reduce factors by using a dedicated VM for the scalability analysis. During the scalability analysis, we assigned dedicated process and memory resources. This should reduce the effects of other VMs on the scalability server. Furthermore, we repeated each scaling step 5 times and calculated the average to avoid further outliers or scheduling problems. Therefore, we assume the risk of external interference to be low.

External Validity Here, we discuss the generalisability of our results. As discussed in the previous threat to validity section, using scenarios comes with the drawback of specialized results. These could not be transferable to other scenarios. Similar to the other cases, we consider the risk to be low because of our usage of external scenarios. Most of the scenarios are also used for the attack propagation analysis. Therefore, at least some generalisability between the analyses is given. In addition, two scenarios are based on real-world breaches. Therefore, showing to some extent real-world properties. The *TravelPlanner* scenario is based on a common research scenario, and the properties are derived from OWASP. The *Maintenance* scenario is based on a scenario described by industrial partners and extended with security properties from OWASP and NVD. The last scenario is based on a research example from a related approach [10]. Based on their origin, they represent different areas, such as real-world properties and interesting research areas. In addition, these scenarios are also targeting different application domains, such as Industry 4.0, enterprise business systems or cloud systems. Therefore, we assume this risk to be low. Nevertheless, our investigated scenarios focus more on the functional accuracy of the analysis than the general accuracy. The main problem is the missing detailed information about real-world breaches, similar like for **G5**. However, based on our good results, we assume that the general results might still be good. The same hold for our effort reduction goal, which also depends on the investigated scenarios.

We investigate the scalability goal **G7** by generalising the runtime behaviour for different input sizes of `ResourceContainers`. However, this generalisation might be problematic because the other architectural elements might have a different runtime behaviour. Overall, the results for other architectural elements might slightly differ, but the tendencies should stay the same. Regarding the evaluation question **Q7.1**, the other architectural elements also generate the same node type. Only the generated edges might be different. The overall process for creating the edges is very similar. Therefore, we assume that the runtime is similar. Regarding the evaluation question **Q7.2**, the answer is similar. In this state, the graph does not differ anymore between different architectural element types but only has nodes and edges. Therefore, mostly the number of nodes may affect the runtime. However, all the architectural elements produce similar nodes if vulnerable. Hence, we assume the risk to be low.

Construct Validity This section discusses whether the investigated property is appropriate for the investigated goal. Our investigated properties are the used metrics. In other words, this section is about the appropriateness of the metrics for the evaluation goals. We used a GQM plan to reduce this threat, which illustrates the relationship between the evaluation goals and metrics. This clear structure can help to find meaningful metrics for the evaluation goals. In the following, we will discuss the appropriateness of the metrics for each evaluation goal.

The first goal **G5** is investigated by the evaluation question **Q5**. For answering the evaluation question **Q5**, we use the metrics precision, recall and F1. They give us insights into the relation between true positives, false positives, and false negatives. Therefore, they give experts insights into the accuracy of the results and what accuracy they could expect. Similar insights can be found in approaches like Seifermann [166] or Pilipchuk [143]. Also, our existing work, such as the attack propagation analysis [211] or Walter et al. [210], use identical metrics to answer the accuracy.

The evaluation question **Q6** regarding the effort reduction goal **G6** is answered similarly to **Q3.1** by a counting metric. It is a simple ratio between architectural elements and is used in other approaches, such as Busch [36] or in our attack propagation analysis [211]. In the previous analysis, the metric $M2.1$ is also investigated for the effort reduction. In this evaluation, we did not consider it. The metric calculates the effort for the next potential propagation step. This is useful because, during a manual analysis of the attack propagation, the security expert does not know the end of the propagation. Hence, they need to manually look at the next connecting elements and determine whether the propagation is possible or not. In our targeted attack graph analysis, this is not the case since the end is already predetermined by the targeted element. Therefore, we did not consider the metric here.

In terms of the scalability goal **G7**, we address evaluation questions **Q7.1** and **Q7.2** by analysing the runtime in relation to the scaled `ResourceContainers`. This means we analyse the runtime behaviour with different input sizes. This behaviour is very common for analysing the scalability and is also performed in similar model-driven approaches, such as Heinrich [70] or related attack propagation approaches, such as Polatidis et al. [148]. Hence, we assume that the chosen metric is appropriate for the evaluation goal.

Reliability This section is about whether other researchers can reproduce our results. All our evaluation questions are answered by metrics. The usage of metrics can avoid subjective interpretation by others and, therefore, increase reproducibility. In addition, by using the GQM approach for our evaluation, we provide a clear structured layout between the investigated goal and the metrics. This can also increase the comprehensibility of the evaluation and increase the reproducibility since it makes for other researchers clearer what metrics contribute to what goal. Furthermore, we provide a dataset [208] containing all the input models, source code, and binaries to execute our analysis. This enables other researchers to easily verify our results or even build up on the approach itself.

The reproduction is also possible on different hardware as the used hardware for our evaluation. The results of the evaluation goals **G5** and **G6** are independent of the used hardware. They only require a compatible processor technology (ARM, x86) and an operating system supporting *Java* 17 with *Eclipse*. The results for the scalability **G7** might change depending on the concrete used hardware, but the drawn conclusion should be similar. However, changing the core numbers for **Q7.1** might drastically affect the results since the application is multi-threaded. However, for the **Q7.2**, this should have no effect because the finding of one attack path is only a single thread.

7.4. Assumption and Limitations

In this section, we discuss the assumptions and limitations of our approach. We choose a combined description because some assumptions result in limitations, and some limitations induce the usage of an assumption. In our case, we will always first give the name of the assumption or limitation and then describe them in more detail. In addition, we provide some ideas on how to relax or circumvent them.

Third Party Interaction Certain types of attacks require the involvement of a third party who is not the actual attacker. This third party is typically a regular user or a malicious internal collaborator. Currently, we are unable to distinguish between these types of attacks. While we integrated the CVSS description into our metamodel, enabling us to differentiate vulnerabilities based on whether they require actions from third parties, such as clicking a

button, we do not support the differentiation in our attack analyses. For our attack analyses, we assume that the third party interaction is automatically given. Here, a potential solution can be the consideration of the usage scenarios in PCM. In these, the third party behaviour can be specified within a usage scenario and then considered during the attack propagation. However, realizing this functionality also requires probably some metamodel changes since a matching between vulnerabilities and usage scenarios is required.

Consistency between Model and System Our approach assumes the existence of current architecture models for the systems, but this is not always the case. While it is beneficial to design the software architecture and integrate our design time analysis during the design phase, legacy systems may not have architecture models. However, existing reengineering approaches like SoMoX [25] can help generate these models and reduce the effort, but some manual effort is still required. Moreover, the architectural model must be consistent with the actual system. Otherwise, our analysis might produce incorrect results. To address this, we can integrate approaches, such as Monschein et al. [119] or iObserve [70], which propose consistency algorithms between design time models and runtime artefacts.

In addition to generating and updating the architectural model, the vulnerability model must also be updated. New vulnerabilities can arise for the system's components during its lifetime, and not all vulnerabilities are automatically fixed and updated. Therefore, it is beneficial for our analysis to consider the latest vulnerabilities in the system. The first steps in integrating these vulnerabilities are already done with the development of the automatic extracting approach in Kirschner et al. [94].

Software Vulnerabilities Our vulnerability metamodel is closely related to the concepts of Common Vulnerabilities and Exposure (CVE), Common Weakness Enumeration (CWE), and Common Vulnerability Scoring System (CVSS). While this brings some benefits regarding the reusability of existing vulnerability knowledge, it also comes with drawbacks and limitations. The vulnerability metamodel is limited to software vulnerabilities and only supports to some extent physical vulnerabilities. The latter one covered only very closely linked to the software. For instance, the Spectre [95] vulnerability is assigned, among others, the CVE-2017-5715 and the CWE class CWE-203 [127]. Therefore, to some extent, physical vulnerabilities can be expressed.

However, other attacks like *Social Engineering* [140], where attackers try to exploit humans to get access or *Invasive Attacks* [198], which directly modifies the hardware, are not considered. Here, a possible solution could be the composition with other attack analyses which specialise in other attack types. For instance, Kramer et al. [99] provides an approach for analysing physical attacks on software architectures, which could be coupled by using coupling approaches similar to the described source code and architecture analysis coupling in Schulz et al. [163].

Besides the neglect of certain attack types, also the concrete classification of matching attack types is limiting. For instance, the CVSS scoring is often considered to be difficult to interpret, and it does not describe the impact well [185]. In our case, the scoring is not problematic since we only use the base metrics. However, there the classification can also be very subjective, depending on the classifier, for instance, for the assignment of the attack complexity. This also covers the classification of CVEs to CWEs. For instance, the vulnerability CVE-2021-28374 [129], used in our running example, was reclassified after one year to a different CWE. Another point is that by using CVEs, we only limit the approach to known attacks, which we will discuss later.

Known Attacks Our attack analyses require that the vulnerability or attack is known. While it does not have to be a concrete vulnerability like a CVE, the approach requires the attack type (here CWE) to be known. Our approach cannot be used to detect unknown vulnerabilities in a system. Nonetheless, our approach can still be valuable in identifying unknown vulnerability chains [46]. These are the exploitation of multiple vulnerabilities by an attacker. Our analyses can be used to create "what-if" scenarios to address the short-coming a bit. In these scenarios, experienced software architects or security experts can assign common weaknesses by using CWEs to architectural elements and investigate the potential impact. Such common weaknesses can be extracted from online sources, such as the OWASP 10 [137], which also contain the corresponding CWE classes. These scenarios can then be used to prepare in case there is an actual vulnerability discovered. Nevertheless, the usage of "what-if" scenarios still does not remove the limitation of not finding new vulnerabilities. This problem can be traced back to the uncertainty about vulnerabilities in the system.

Uncertainty about Attackers Another assumption of our approach is the knowledge about the attackers. We assume that security experts can model the capabilities and knowledge of attackers. However, in reality, this is often uncertain. Especially for our attack propagation analysis (C4.1), this knowledge is important because we require the concrete capabilities, knowledge and starting points. In the targeted attack path analysis (C4.2), we allow more uncertainty because we only restrict the solution space by attack filters. Nevertheless, it still requires at least the knowledge for the filters. In Walter et al. [209], we already consider some mitigation techniques for uncertainty. There, we created a variation model and combined it with our attack propagation analysis to evaluate the attack propagation for different starting points. However, the approach comes also with additional drawbacks, such as a high-performance overhead. Here, the combination with other uncertainty mitigation approaches, such as Hahner et al. [66], might be helpful.

Consideration of other Security Properties Our vulnerability metamodel considers similar to CVSS the impact on confidentiality, availability and integrity for a vulnerability. Our attack analyses consider the confidentiality only by using the parameters. The other security properties are not considered in the attack analyses. Nevertheless, these properties can be relevant. For instance, in an integrity attack, attackers could modify components to use other credentials and thereby gain access to them. This also includes the differentiation for writing or reading operations in our access control metamodel. So far, we do not consider this explicitly.

States of access The attack analyses need to differentiate between accessible and compromised architectural elements. The difference is that in the first case, the attacker can only access the architectural element as a regular user. In the second case, the attacker has compromised the architectural element and has full control over it. In our attack propagation, we already have the first steps for a differentiation between reachable services and confidentiality threats. There, attacks can read data from services which are vulnerable. However, this does not propagate further to other services which are called. Furthermore, the targeted attack path analysis does not have this. Here, a finer differentiation could be useful. For instance, by having two types of nodes in the attack graph. One for accessible architectural elements and one for compromised or full control. In this regard, also a finer differentiation

between different access layers can be considered. Currently, we assume that if attackers get access by using credentials, they can fully control the architectural element. However, it might be beneficial to differentiate here between privileged access and non-privileged access. In our case, we modelled this differentiation by assigning access rights to services and the usually more privileged maintenance access to the component. Another alternative is to assign this to infrastructure services or add a second layer for services. The first might be easier because the latter would require changes in the used ADL.

Implicit Attacks Most of the attackers' capabilities are modelled explicitly, meaning that we have to specify them and assign attacks to attackers. In contrast, we assume that attackers can perform some attacks implicitly. Our first implicit attack is the automatic propagation from `ResourceContainer` to the `AssemblyContexts` deployed on it. The underlying assumption is that compromised hardware or operating system automatically affect the executed programs. This assumption is valid for most cases, but not all attackers might have the capability, or there might be mitigation steps to prevent the attack. Nevertheless, we choose to keep this assumption because it simplifies the propagation rules and still holds in most cases.

Another implicit assumption about the attackers' capabilities is the usage of the attributes as credentials. We assume that an attacker automatically has the capability to use every gained attribute in the system. In reality, this might not be possible. However, we chose this assumption since it simplifies the analysis and is comparable to other related access control analyses.

Advanced Mitigation Our approach only supports simple mitigation strategies like access control and network segregation and does not consider advanced strategies like data encryption or trusted execution environments [158]. Nonetheless, our attack analyses can be used to identify potential mitigation locations. For example, security experts can analyse the system and get potential attack paths and propagations. Based on these results, the security experts can determine architectural elements which are critical and often affected by attack paths. Security experts can then introduce mitigation strategies to break the attack paths for these elements.

One strategy to handle the limitation of the mitigation mechanism is the manual removal of mitigated vulnerabilities by security experts or architects. After the removal, they can reanalyse the system and thereby analyse a software architecture with mitigations in it. However, this would lead to an inconsistent model regarding the vulnerabilities, and it is a high manual effort.

Furthermore, these mitigation strategies can be compromised in practice [160]. Therefore, they should be considered during the attack analyses. At least the mitigation strategies should be made explicit so that they can be manually verified or automatically with approaches, such as Taspolatoglu et al. [195].

Handling of Data The data model in our analyses is very limited and only considered from the control flow. The access control rules are only defined on services, components, and devices. Data is only indirectly considered. There are certain ways to circumvent this, like in our education evaluation scenario (c.f. Section 6.6), where we defined a component for a data object and assigned operations for it. However, this is more like a workaround than a good modelling solution.

In combination with the limited mitigation model, this limited data model also limits the consideration of encrypted data during the data extraction in the attack propagation. The data extraction does also not consider concrete data instances. Hence, it cannot be used to differentiate data in multi-tenant scenarios. The attack propagation analysis can only provide that all data from a certain type is affected. In addition, the attack propagation analysis cannot extract data from possible data flows in the system. In detail, in a real system, a compromised network element could potentially see all the data which is exchanged over it. However, we do not consider the data flow in our analysis. Therefore, this data is not considered in our analysis. Furthermore, the targeted attack graph analysis does not even provide the data extraction feature so far.

Scalability for Large Software Architectures As the evaluation of our attack analyses showed, there can be a performance issue for larger software architectures. This performance issue can limit the application of our analyses. However, in the case of the targeted attack path analysis, software architects

or security experts can influence the performance by using a filter. Using filters, such as the `StartFilter`, can reduce the problem size. Therefore, it can speed up the analysis. The drawback is that it might affect the accuracy of the results because the filter removes potential attack paths. Nevertheless, the drawback can be justifiable in some cases, for instance, by considering different trust levels in the architecture. There might be elements in the software architecture, which are more vulnerable and less trustworthy than others, such as the externally accessible components like in running example the `Terminal` or outdated components. Nevertheless, for larger software architectures, the filters are also useful, because they restrict the number of results. For instance, in a software architecture with around 1000 elements, our approach can calculate the results in around 17 minutes. However, in the worst case, the result would also contain around 1,000 attack paths with up to 1,000 elements per attack path. This might be already too many results to handle for security experts. Hence, they must filter the results to identify the relevant attack paths.

Access Control Model XACML is the foundation for our access control metamodel. However, our analyses do not support its full functionality of it. The most restricted analysis regarding the access control model is the targeted attack path analysis (C4.2). Currently, it only supports the label comparison of string attributes.

For the attack propagation, the limitation is regarding the attribute selection. During the access request creation, the analysis creates a bag of values and adds all the attributes it has in the bag. The bag is then used by the PDP to extract the attributes and compare whether they match. Here, two problems can arise. We explain the first example by giving a short simplified example. An exemplary access control policy could require the requestor to have the attribute a but not the attribute b. If our analysis gained both attributes, it would add a and b to the bag of values. The PDP would then deny the access because we also have the b attribute. However, a malicious attacker could just simply not send b and then would get access. This tempering of the attributes would gain them access. This problem is called attribute-hiding attack [42]. Our analysis does not consider this issue and assumes that no such policies exist. A solution for the problem is the usage of dedicated policy analyses, such as Turkmen et al. [202] for identifying attribute-hiding attacks. This can be done, for instance, by using our generated XACML file.

The second problem is with the bag itself, but it can also be traced back to attribute tampering or hiding. For instance, an exemplary access control policy could say that the access is granted if the first element in the bag is *a*. Our analysis does not consider such orders in the bag creations. Therefore, the value *a* might not be at the correct position in the bag, despite the fact that the attacker has the value. A real attacker could try to change the order in the bag. However, our analysis does not. Here, we also assumed that our access control policies do not use this.

This assumption is also used in access usage analysis. There, we have the same problem, that the order of the bag is determined during the request creation and cannot be specified in the model.

Also, the XACML transformation does not support every available element in the metamodel. Especially, the different datatypes cannot be used. Currently, only the string datatype is supported in the implementation. In addition, the model editors are also only considering string serializations. However, both aspects can be solved by adding datatype converters to the transformations and model editors.

Also regarding the XACML transformation and the possible reusability of the generated policies during the runtime, we assume that element identifiers are identical to the identifiers in the architectural model. During the transformation and the analyses, we use the PCM identifiers. However, in the real system, these might vary. In this case, the PDP cannot identify the correct element and might deny a request.

Static Access Control Policies As described in our problem definition (c.f. Section 1.2 — P1), our access control policies are not self-adapting to a new environment. Hence, the policy itself is static, while the access decision is dynamic based on the context. In very dynamic environments, where the access control system needs to automatically update to the new situation, such as in Bureš et al. [34], our approach cannot directly be used. However, suppose the adaption does not happen too often, meaning that the policy stays the same for certain periods, for instance, because the situation does not change, the current state of the adapted policy can be extracted as a static policy. This policy can then be used as input for the analyses.

Potential Overestimation of Attacker Capabilities Our metamodel simplifies the attackers' capabilities to exploit vulnerabilities based on the identifier of CWEs and CVEs. While we already consider some additional factors, such as access control and the attack vectors, other factors might not be considered. For instance, a static code analysis might find a potential weakness (CWE) within a component. However, the existence of a weakness does not automatically mean that there is an actual exploitation for attackers available. Nevertheless, if we model it based on the static analysis, our analysis would assume that attackers can exploit it. A similar issue is that we assume that every CVE can be compromised by its CWE class. In general, while the weakness is the same, the actual attack might be so different that an attacker might not know the exploitation. In our evaluation scenarios, we did not see this behaviour. However, this could be a limitation in other scenarios. Also, other aspects, such as required resources like time or money, are not considered. Other works, such as Yoshizawa et al. [222] or Ponikwar et al. [149] differentiate between the motivation and attacker types such as state (in the sense of a nation) attackers or "script kiddie" (amateur hacker). In our work, we can only do this indirectly by assigning the CVEs.

Supported Architectural Elements On the technical side, we only consider `AssemblyContexts`, `Services`, `LinkingResources`, and `ResourceContainers` for the attack propagation. For the targeted attack path analysis, we support the same elements without the `Services`. The access usage analysis only considers the `Services`. Specifically, the support is only for `Services` based on `OperationInterfaces`. Services in other interfaces, such as the `InfrastructureInterface`, are not considered. However, our chosen elements are the most commonly used elements in different PCM analyses. Nevertheless, in the future, the analyses might be extended there to support more elements. In most cases, the extension should be possible since it only requires new propagation rules for the new elements.

7.5. Overall Evaluation Results & Discussion

In this section, we summarize the evaluation results for our different analyses and discuss how well our contributions answer our research questions based on our findings in the evaluation.

Our first research question **RQ1** covers the aspect of access control policies. In detail, the sub-question **RQ1.1** is answered by the access control metamodel, which is the contribution C1. In our evaluation, we do not investigate the quality of the contributions independently of our second sub-research question **RQ1.2**. This research question is answered by our contribution C3. The contribution consists of the access usage analysis, which analyses the access control policies based on modelled usage scenarios. These usage scenarios contain context-based attributes and represent a context-dependent scenario. In addition, we used the access control model (C1) to represent access control policies. Using the context-dependent scenarios, we can statically analyse the system regarding potential violations as described in **RQ1**. We evaluated the contribution C3 with our evaluation goal **G1** regarding the accuracy of analysis results. Because the contribution uses internally also the contribution C1, we indirectly also evaluate the pragmatics of the metamodel. This indirect evaluation is based on the fact that the analysis uses the metamodel. Therefore, if the metamodel, for instance, is missing elements which are required for the analysis, the result is affected. This results in that if the metamodel is bad for modelling access control policies, our evaluation results for C3 would be affected.

For the evaluation, we used four different evaluation scenarios with overall 18 different usage or misusage scenarios. During the evaluation, we investigated two evaluation questions, which focus on the correct access decision for services and the correct access decision for a usage or misusage scenario. In the investigated scenarios, we had cases with access violation and non-violation. We also had usage and misusage scenarios. Therefore, we investigated a wide variety of scenarios for our analysis. We compared the analysis results against a manually derived reference output. The evaluation suggests a high accuracy. Our analysis provided a JC of 1.0 for our scenarios, meaning that all the analysis results match the expected results. Based on the good evaluation results for our contribution C3, we can assume that, to some extent, the results are also based on the quality of the metamodel. Therefore, the good evaluation results indicate to some extent a good quality for the contribution C1. Overall, the evaluation indicates that our answer with the access control metamodel (C1) and the access usage analysis C3 for the research question **RQ1** have a high accuracy and therefore are good solutions.

The second research question **RQ2** investigates the propagation of attackers in a software architecture. The first sub-research question **RQ2.1** investigates the relevant properties of an attack propagation. We answer it with our con-

tribution C4 and C1. The first contains the metamodel for the vulnerabilities, the attacks, and the attacker. The second contains the access control metamodel because attackers do not only propagate by exploiting vulnerabilities but also exploit existing access control policies. As for the research question **RQ2.1**, we do not explicitly investigate the quality of our contribution in our evaluation but rather investigate them with our second sub-research question **RQ2.2**. The reason is the same as for the C1, that we indirectly evaluate the pragmatics by using the metamodels in our analyses. The research question **RQ2.2** investigates the different attack analyses. We answer it with our contribution C4. This contribution is split into two sub-contributions. The attack propagation analysis is contribution C4.1, and the targeted attack path analysis is contribution C4.2. Both attack analyses reuse the metamodels developed in the contributions C4 and C1. Therefore, these contributions affect the results of the attack analyses because a non-appropriate metamodel might result in bad analysis results.

The attack propagation analysis, the contribution C4.1, is evaluated regarding the evaluation goals accuracy **G2**, effort reduction **G3**, and scalability **G4**. For **G2** and **G3**, we investigated 18 different evaluation scenarios and sub-scenarios. The scenarios are built upon real-world breaches and evaluation cases enhanced with security vulnerabilities from security advisors and vulnerability databases. For the evaluation goal **G2**, we manually created a reference set containing the expected attack propagation, the expected affected data, and attack complexity. We compared each reference set with the metrics precision, recall, and F1 to the analysis results. In all investigated evaluation scenarios, we got a precision, recall and F1 from 1.00, which is a very good result. For the effort reduction, we got mixed results depending on the evaluation scenario. In some evaluation scenarios, where the attack propagation only affects a part of the software architecture, the results are quite high. In other evaluation scenarios, where nearly the complete system is affected, the effort reduction is lower. In addition, we discussed the effort for applying our approach and showed potential approaches to reduce the effort further. For investigating the scalability (**G4**), we scaled an input model regarding `ResourceContainers` and measured the runtime. The results showed a potential long runtime for software architectures above 10,000 elements. However, we identified different possibilities to improve the performance. Overall the evaluation results indicate that our contribution C4.1 answers research question **RQ2.2** satisfactorily.

We evaluated the targeted attack path analysis, the contribution C4.2, regarding the evaluation goals accuracy **G5**, effort reduction **G6**, and scalability **G7**. During the evaluation goals **G5** and **G6**, we investigated five evaluation scenarios with 52 potential attack paths. Our evaluation scenarios stem from real-world breaches and evaluation cases. For specifying the vulnerabilities in the evaluation scenarios, we used security advisors and vulnerability databases. Based on the evaluation scenarios, we manually checked the possible attack paths for each scenario. Afterwards, we compared in **G5** the found attack paths and their validity with the metrics precision, recall, and F1. For all evaluation scenarios, we have a precision of 1.00, meaning that all found attack paths are valid attacks. In two evaluation scenarios, we missed some attack paths, which is why our lowest recall is 0.86 and our lowest F1 score is 0.93. Nevertheless, these results are quite good and indicate a high accuracy. For the effort reduction, we again get different values depending on the actual attack path. For the scalability goal **G7**, we investigated the runtime with scaled `ResourceContainers` for the graph creation and the path identification. Here, the results are better than for the attack propagation analysis. The runtime for each part with 100,000 elements takes around 1.5 hours. Nevertheless, we also could identify some performance improvements. Overall the good evaluation results indicate that our contribution C4.2 is satisfactory. In combination with the evaluation of the contribution C4.1, the contributions C4.2 provide in our eyes a sufficient answer to the research question **RQ2.2**.

Both analyses together are our contribution C4 and answer research question **RQ2.2**. The evaluation results indicate that the contribution answers the research questions well. Furthermore, based on the evaluation results, we can assume that the contribution C2 and C1 answer the research question **RQ2.1** satisfactorily. This is based on the fact that our attack analyses use the metamodels developed in the contributions C2 and C1. Therefore, if the metamodels are not sufficiently modelled, the analysis might result in faulty results. However, most of the results for the attack analysis are good. Nevertheless, there are some non-ideal results. The bad scaling behaviour is influenced by the metamodel. The problem is less with the attributes used in the metamodel but more related to the technical realization and underlying framework because they require repeatable searches in lists of elements to identify the relevant model element. Hence, the problem is the used extension mechanism and not the metamodel itself. The other open problem regarding the missed attack paths in **G5** is not based on the metamodel but rather on

the used path finding algorithm. Hence, we assume that our contributions C2 and C1 sufficiently answer the research question **RQ2.1**. Overall, based on our evaluation, we can say that we have answered our research questions **RQ2.2** and **RQ2.1** sufficiently. As a result, we also assume that we answered the research question **RQ2** because it is composed of the research questions **RQ2.1** and **RQ2.2**.

Part IV.

Epilogue

8. Related Work

This chapter discusses related approaches and tools to our developed approaches. We split the related work along our two main research questions **RQ1** and **RQ2**. The first Section 8.1 discusses approaches for analysing and modelling confidentiality in software systems. This covers, among others, different access control models and their analyses. In this section, we focus on approaches with no explicit or no variable attacker models. Approaches containing these properties are described in Section 8.2. Here, we focus on approaches describing attackers, their behaviour, their attacks, and vulnerabilities in the system. In addition, this contains different analyses using the descriptions.

8.1. Approaches focused on Confidentiality

This section discusses different approaches to analyse confidentiality. As previously described, confidentiality is often provided by the access control system. Other approaches, especially in combination with privacy aspects, focus on anonymising the data, such as K-Anonymity [194] or Differential Privacy [51]. Therefore, they provide means to protect the identity of the data provider and, therefore, try to preserve the confidentiality of the data provider. Another possibility to protect data from unauthorised access is the usage of encryption. In our work, we focussed on the access control and similar related approaches and analysis. Before discussing different analyses, we first discuss in Section 8.1.1 different access control models in contrast to our chosen model, which is ABAC. Afterwards, we describe different access control policy analyses in Section 8.1.2. Here, we focus on approaches which use existing policies and analyse them. We discuss various confidentiality analyses in Section 8.1.3. Here, we focus on model-driven analyses. Usually, access control only works at the moment data is accessed. After the data is accessed, it might be that the data left the original data dominion, and

theoretically, the data is unprotected. For solving this problem, usage control approaches are used. We will discuss these in Section 8.1.4. After introducing different scientific or applied research approaches, we will discuss some exemplary industrial tools in Section 8.1.5.

8.1.1. Access Control Models

In our approach, we chose to use ABAC as our access control model because it provides support for considering the context during the access control decision. There exist various other access control models [118]. Some of these can consider more attributes of the context others can consider less. The first access control model is Discretionary Access Control (DAC) [205]. It defines access control policies on the user level and only uses the user's identity to determine access. Therefore, the formalization of policies is complex since access needs to be granted to single users. In addition, it does not consider additional attributes like, for instance, the state of the machine in our running example. Another approach is Mandatory Access Control (MAC) [206]. Here, the access is controlled by a central entity. It differentiates access between persons and processes and classifies the requested object, for instance, in *secret* and *public*. A popular representative is the SELinux system to restrict users in a Linux system[1,2]. Still, the definition can be quite complex, and there is no possibility of considering different context states.

In contrast, RBAC [55] abstracts from the user by grouping similar users and activities in roles. The access is then defined on the roles, not the user level. For instance, the technician has the role technician in our running example. This helps in the formalization and in the comprehension because the policies abstract from the individual user performing a task to the role performing a task. This role assignment can be seen as a first step to considering the context. However, the context consideration is very limited because it only covers the role and other aspects, such as the state of the machine in our running example, are not considered. Several approaches exist extending the RBAC approach to address some of the shortcomings. The dynamic RBAC

[1] *National Security Agency | Central Security Service > What We Do > Research > SE Linux.* URL: `https://www.nsa.gov/what-we-do/research/selinux/` (visited on 09/09/2019).

[2] *SELinux Userspace.* URL: `https://github.com/SELinuxProject/selinux` (visited on 03/20/2023).

(DRAC) [225] approach automatically updates the role assignments for users based on the context. Therefore, it can consider further context attributes. Nevertheless, the access is still determined solely by roles, and the context cannot be directly used. A similar concept is used by temporal RBAC (T-RBAC) [28], where roles are enabled and disabled based on time constraints. However, there are also approaches which add additional context attributes to the policy itself. For instance, privacy-aware RBAC [124] adds the context of privacy-relevant information, such as purpose and obligations to RBAC. Therefore, enabling a more privacy-aware access control. The same can be done for ABAC, for instance, by using the dedicated extension points by adding attributes and custom comparison methods. Yet, ABAC still provides the consideration of more context attributes.

The Role-centric attribute based access control [83] aims to combine the advantages of RBAC and ABAC. It assigns attributes to the requestor and the requested object. However, in general, in contrast to ABAC, it does not consider the environmental context.

The Organisation-based access control (OrBAC) [88] tries to tangle some of the limitations of RBAC. It was developed for managing access within an organisation and considers multiple different contexts, such as time or user-related properties [43]. Many extensions exist for special purposes, such as Trust-orBAC for considering trust [197] or special extensions for cloud computing [221]. The approach can be an ideal candidate as an alternative to our used ABAC. The approach could be integrated by replacing the PDP and the access control query and transformation modules with modules for OrBAC. However, in our approach, we choose ABAC together with XACML because, in our eyes, they provide better documentation, more freely available tools, and industrial applications.

A recent trend in access control models is the consideration of risk in the access decision. It is some kind of dynamic access control approach targeting dynamic environments, such as IoT. There exist various different risk-based access control models [16]. They use different properties, such as the context, the history, or the sensitivity to determine the access decision with approaches, such as Fuzzy Logic, Machine Learning, or Game Theory [16]. Based on the diversity of the considered properties, the approach itself is comparable to ABAC. In our case, we choose to use ABAC because of its maturity and better tool support.

The Unified Access Control Modeling Language (UACML) [180] aims to find a common metamodel for all access control approaches. It is based on UML and supports the expression of access control requirements independent of the chosen access control model. This metamodel can then be integrated into existing design time modelling approaches. In contrast, to our approach, their approach is more generic and includes multiple access control models, such as DAC and MAC. However, they do not support attribute access control so far, and there are no analysis capabilities.

Another model-based approach is SecureUML [106]. It extends UML with the possibility to model access control properties and generate, based on the model system, the runtime policies. Therefore, they are very similar to our access control model and the transformation to XACML, where we also can generate runtime policies. Our approaches differ in the chosen access control model. Within their approach, they mostly use RBAC, but they extended it with the capability to express dynamic expressions. These dynamic expressions are realised by using OCL constraints. Therefore, they can express situations like in our running example, that access is granted only during a certain state of the machine. In contrast, our approach uses ABAC, an approach which natively can already support different context attributes. Furthermore, our access control model is based on an existing industrial standard for modelling access control, which can increase the applicability for practitioners. This is also continued in the generated access control policies. SecureUML generates access control policies for EJBs (Java Enterprise Beans), which are usually only used in the Java Enterprise domain. In contrast, our generated access control policies are XACML policies. Despite the requirement of an XACML-based PDP, these policies can be used in heterogeneous programming language applications, and it is additionally a common foundation for access control policies and their analyses.

8.1.2. Access Control Policy Analyses

Jabal et al. [80] provides an overview of different policy analysis approaches. They investigate not only access control policy analyses but also consider other analysis types, such as network policies or SELinux policies. Kasten [89] also discusses different policy models, such as flow control or usage control.

Alberti et al. [9] provides a policy analysis approach for analysing an extended RBAC model. Their used access control model supports the consideration of additional contexts. For analysing the policy, they use a symbolic model checking approach. They analyse the problem arising through role delegations. In other words, in RBAC, a role can delegate its permission to another role. In contrast, in our approach, we do not support role delegation because it is not included in our metamodel. However, there exist an extension for XACML supporting delegation [133]. Using the extensions and slightly adapting the propagation rules, our analysis might provide similar results. Nevertheless, their approach is based only on the policy level, and our approach uses the software architecture. Therefore, our approach needs the software architecture but can in return identify components which are problematic.

One policy analysis for XACML-based policies is Margrave [57]. It uses decision-diagrams [58] internally to calculate whether users can perform certain actions and get the impact of access control policy changes. In contrast, our contribution C3 focuses more on a scenario-based testing approach. Turkmen et al. [202] introduces also a XACML-based policy analysis. They use satisfiability modulo theories (SMT) to analyse different access control policies regarding attribute hiding and change impact. Other similar approaches are Hughes et al. [75] using a sat solver for the analysis, Lin et al. [104] checking for similarities, such as effect, Jabal et al. [79] determining various quality attributes for policies or Ait El Hadj et al. [7] clustering policies to detect anomalies, such as redundancy or conflicts.

Overall, our approach differs from the mentioned one by focusing on the software architecture. The mentioned approaches consider mainly only the policy and do not consider further information, such as components or hardware devices. Therefore, our approach does need more initial input. In return, we provide additional feedback about architectural elements and, in the case of C3, additional about the intended usage. In contrast, our feedback about access control policies is more limited. Furthermore, our approaches in C4 consider, besides the access control policies, additionally a dedicated attacker model with vulnerabilities and attacks. This can provide feedback on how attackers can exploit access control policies in combination with vulnerabilities. Nevertheless, it might be beneficial for security experts to analyse our generated access control policies with other XACML-based analyses to get insights into other security properties. This is possible because security experts can use our generated XACML file. Therefore, the XACML file acts as

a universal exchange format. On the technical side, there might be a problem if the analysis uses different XACML versions, but there should be no other restrictions.

Besides the analysis of a policy regarding certain security or quality properties, there exist also approaches that focus on the adaption of these policies regarding a dynamic environment. Silva et al. [175] is such a self-adapting policy model. It actively analyses the user behaviour with Markov-Chains and can grant or remove permission for users. In contrast, in our approach, we explicitly excluded the adaption of the policy itself. However, it might be possible to integrate such approaches as long they can handle XACML policies.

8.1.3. Confidentiality Analyses

In this section, we focus on confidentiality based on access control, especially if they exploit model-driven techniques. Different surveys, such as Den Berghe et al. [49] or Nguyen et al. [123], identified that many security-related model-driven analyses target access control or confidentiality.

Zhang et al. [226] provides a very similar approach to C3 and C1. They developed a custom Domain Specific Language (DSL) for specifying access control policies. They later transformed the access control policies specified in their DSL to a XACML-based policy. In addition, security experts can define goals and malicious goals. These are operations a user should be able to perform. Afterwards, they check whether a user can perform these. This is comparable to our misusage and usage scenarios in C3. Our approaches differ in that we additionally use the software architecture and can consider service calls which are delegated by the original service call. In other words, services can call other protected services, which can have different access control policies and might be denied. Therefore, we have a more holistic view of the system. Another exemplary approach using model checking to analyse access control policies is, for instance, Guelev et al. [63].

In Basin et al. [21], SecureUML [106] is extended for a policy analysis. The authors combine the security model and the component level to enable OCL queries regarding the access control policies. Exemplary queries are, for instance, the identification of required roles for accessing an object or whether two rules have the same permissions. In contrast, in our work for C3, we

focussed on a given scenario and for C4 on the propagation of attackers, which are so far not considered in SecureUML.

Sectet [64] is a model-based access control framework designed for workflows. It allows the modelling of workflows with security properties. Like our approach, Sectet transforms the modelled access control properties into XACML files, which can then be used by PDPs to determine the access decision. In contrast to our approach, Sectet focuses on workflow definitions and does not take into account the software architecture aspect. In addition, they do not consider vulnerabilities together with attack propagations, which is a key aspect of our work.

The UML-based Web Engineering (UWE) [37] extends UML for modelling different security properties. It also provides support for generating access control policies. In Bertolino et al. [30, 31] UWE was used as a foundation for test case generations for XACML. These test cases can be seen as similar to our scenarios. However, in contrast, our analysis also supports attack propagation.

The Data-centric Palladio [167, 168] approach extends PCM to support data-flow-based confidentiality analyses. It mainly supports access control and information flow analyses. The idea is to describe data with characteristics. These can contain role information or other attributes. At each data processing node then, the characteristics of the data can be checked against the required characteristics of the data processing nodes. In addition, there exist Prolog queries, which define the access control or information flow policies. There exist, also other extensions like our context-based access control model [33]. Other extensions consider uncertainty in the input values [32] or handling structural uncertainty [210]. In contrast to the data-flow-based approach, our access control model (c.f C1) is defined on services. Furthermore, we also support the concept of misusage diagrams, which are so far not considered in Data-centric Palladio. In addition, our approach has a dedicated attacker model and uses the access control properties to propagate attacks. Also, our access control model follows more closely the industrial standard and our access control policies can be reused during the runtime. Nevertheless, the data flow analysis might be better in more data-oriented applications than our approach. Also, the SecDFD [200] approach is a data-flow-based analysis, which was extended in Tuma et al. [199] for automatic detecting of security flaws based on the source code. However, they focus on information flow analysis, whereas we focus on access control.

Other information flow analyses are, for instance, the iFlow approach [90, 91] and Gerking et al. [60]. The first extends UML via profiles and then generates the source code of the system based on the modelled system. In addition, it is combined with a verification step to analyse the system for data flow violations. In contrast, to our approach, they target a different security property with information flow as our approach with access control. Also, they choose their own access control model, which is more oriented on RBAC instead of our access control which is built upon an existing standard for ABAC. Furthermore, we support misusage scenarios and provide a dedicated attacker model. The second approach, Gerking et al. [60], extends MechatronicUML [26] and combines it with timed automata [14] to analyse whether the real-time behaviour and message exchange is secured. As with the previously described approach, our approach targets, in contrast, access control, which is a different security property. In addition, our approach has a dedicated attacker model and different propagation analyses. Nevertheless, we do not support the analysis of real-time properties.

8.1.4. Usage Control Approaches

In a dynamic environment like in Industry 4.0, or IoT, it is important that data can be secured even after leaving the company networks. Usage Control approaches target this security issue by defining restrictions on the usage, such as the data cannot be modified or only read once. $UCON_{ABC}$ [138] is a model which allows specifying policies for usage control. In Pretschner et al. [150], a model is proposed to combine the policy specification of UCON with policy enforcement mechanisms like DRM [193]. While usage control is important in dynamic exchanging environments, they are not entirely necessary for our approach. We focus on the system analysis and not on the runtime decision-making or enforcement, where usage control is necessary. However, it might also be possible to integrate the concepts into the policy generation similar to Munier et al. [120], which internally uses OrBAC to specify the access control policies or like Hariri et al. [69] which extended XACML to support UCON.

8.1.5. Industrial Tools & Approaches

Besides the previously presented scientific approaches, different industrial approaches exist considering contexts for the access decision. Axiomatics[3] provides various systems for managing ABAC-based systems. Among other tools, they provide a PDP for XACML. In contrast, our chosen PDP is freely available. Nevertheless, it might be possible to integrate their PDP into our approach. They also developed a custom DSL Alfa[4] for specifying access control policies. In this regard, their DSL is very similar to our contribution C1. In contrast, our metamodel has special elements to support the easier integration into our used ADL.

Regarding the risk-based authentication, various implementations exist, such as Okta[5] or IBM[6].

8.2. Approaches focused on Attacks & Attackers

This section discusses related approaches to our contribution C2 and C4. It also discusses parts of C1 if they are related to parts of the attacker.

8.2.1. Vulnerability & Attack Classifications

In our approach, we model our vulnerabilities based on the classifications of CVE, CWE, and CVSS. These provide most of our properties describing a vulnerability in our metamodel. All three approaches are also commonly used in the industry or vulnerability databases. There also exist other classifications. For instance, Garg et al. [59] proposes a classification based on technical parameters, such as the targeted operating system or techniques. However, their approach also uses internally CVSS for the identification of the severity.

[3] *Axiomatics*. URL: `https://axiomatics.com/` (visited on 03/24/2023).

[4] *Axiomatics releases free plugin for the Eclipse IDE to author XACML 3.0 policies*. URL: `https://axiomatics.com/news/press-releases/axiomatics-releases-free-plugin-for-the-eclipse-ide-to-author-xacml3-0-policies` (visited on 03/24/2023).

[5] *Risk-Based Authentication: What You Need to Consider | Okta*. URL: `https://www.okta.com/identity-101/risk-based-authentication/` (visited on 03/20/2023).

[6] *IBM Risk-Based*. Mar. 7, 2021. URL: `https://www.ibm.com/docs/en/tfim/6.2.2.6?topic=access-overview-risk-based` (visited on 03/20/2023).

Another classification is Li et al. [103]. Both approaches are currently limited in their application, and most databases use a combination of CVEs and CVSS.

Another classification and scoring system similar to CVSS is the CWSS by Mitre [112]. Both approaches are quite similar and consider similar properties. However, they are not identical and target different usages. For instance, CWSS can handle unknown properties for the score calculation. In our case, we do not use the scoring and are more interested in the base metrics. Therefore, we only use CVSS because it is more often used.

Nevertheless, as previously discussed, CVSS has limitations [185, 184]. For some, the most critical aspect is that CVSS is used to determine the risk of a vulnerability, but it only calculates the severity of the vulnerability[7]. For instance, a high severity can be, in some cases, not a problem if the vulnerable component is not critical. This is also in line with our approach (C4), where we also look at the complete system to analyse the impact. For addressing this issue, there exist different initiatives and research. For instance, the Stakeholder-Specific Vulnerability Categorization (SSVC) [186] considers further specialised attributes to calculate a more custom risk. Also, other approaches, such as the Exploit Prediction Scoring System (EPSS) [81] [53], follow a similar approach to consider the risk. However, the usage for both approaches is still very small compared to CVSS, and some even include it.

Regarding the attack modelling, we use in our approach CVEs and CWEs to identify an attack and automatically derive further attributes, such as the attack vector from the context. Another alternative could be the CAPEC [38] by Mitre. It describes textual typical attacks and links different attack types similar to CWE with parents and children. Furthermore, it contains a reference to the related CWE weakness for the attack. In contrast, our model is a bit simpler because it reuses the same concept to model vulnerabilities and attacks. However, because of the usage of CWEs in CAPEC, CAPEC can also easily be integrated into our metamodel and analyses. It would just require adding a new subclass of `CategoryAttack` and overriding the method `canExploit`.

[7] A. Liska. "CVSS Scores Are Dead: Let's Explore 4 Alternatives". Apr. 19, 2021. URL: https://www.rsaconference.com/library/Presentation/USA/2021/cvss-scores-are-dead--lets-explore-4-alternatives (visited on 03/28/2023).

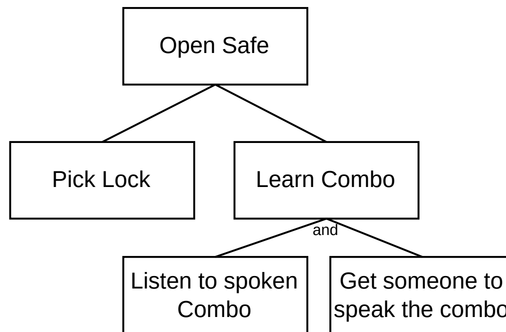


Figure 8.1.: Exemplary simplified attack tree based on Schneier [162]

8.2.2. Attack Path & Threat Modelling

Many attack analysing approaches are modelling attacks as a direct acyclic graph [98, 224, 18, 13]. For instance, Schneier [162] presents attack trees, which are based on fault trees. Figure 8.1 illustrates such an attack tree. For simplicity reason the example is based on the example given in Schneier [162]. It is a simplified example for breaking into a safe. However, the safe in this example could be easily replaced with software architecture elements. The root node describes the goal (here, open safe), and then the sub-goals are described. They can be connected by logical disjunction (*or*), such as for *Pick Lock* and *Learn Combo*, or such in the case of *Listen to spoken combo* and *Get someone to speak the combo* by a logical conjunction (*and*). Furthermore, elements can be marked as impossible if the attack is not possible. In Mauw et al. [109], a formal definition for attack trees is given. Opdahl et al. [135] compares attack trees with misuse cases for threat identification in an experiment. The results were that attack trees have been better for identifying threats. Tøndel et al. [196] presents an approach combining misuse cases and attack trees by linking use cases to attack trees. In contrast, to attack trees, our approach does not explicitly list the impossible sub-goals. However, the analysis results from C4 can be similarly interpreted. For instance, the goal is the target in C4.2, and the subgoals are the necessary architectural elements to reach the target. In the case of the C4.1, the result is not a tree, and it is missing the goal. However, still, the concept of activities leading to compromisation, such as with the subgoals, exists.

A common application of attack trees is during threat modelling [172]. For instance, Alhebaishi et al. [10] use attack trees to identify security metrics in cloud computing during a threat analysis. For the identification of potential threats during the threat modelling, Microsoft developed the STRIDE [96] approach. It stands for 1. Spoofing of user identity 2. Tampering with data 3. Repudiability 4. Information disclosure (privacy breach) 5. Denial of Service (D.o.S.) 6. Elevation of privilege. These also represent the different threat categories. The approach provides a structured framework for thinking and concentrating on security threats and, therefore, can help security experts to identify more threats. Based on this concept, Microsoft developed the Security Development Lifecycle (SDL)[8]. This lifecycle contains a process which should produce more secure and reliable software applications. Also, other processes target this aspect, such as the OWASP Software Assurance Maturity Model (SAMM) [136]. Overall, most of these processes use a manual process for identifying threats. Also, especially the lifecycle approaches are not directly competing with our analyses. In fact, it might be beneficial to integrate our analyses into the mentioned processes.

8.2.3. Attack Path Estimation & Automatic Analysis

In the previous section, we mainly covered the manual analyses and manual generation parts for related approaches. In this part, we focus on the automatic creation of the attack paths or the automatic analysis of graphs.

In recent years, there has been a trend to automate the generation or the automatic analysis of attack trees or attack paths [218]. One keystone for automatic analyses can be the availability of different easy-to-use DSLs for specifying attacker behaviour. The Meta Attack Language [85, 219] tries to tangle this issue by providing a metamodel for DSLs. These can then be used to specify concrete attacker behaviour. Based on this concept, the VehicleLang [92] was developed. It is a DSL for specifying attacker behaviour in vehicles and their IT infrastructure. In contrast, our attacker behaviour is encoded in the metamodel and Java propagation rules.

[8] Microsoft. *Security Development Lifecycle*. URL: `https://www.microsoft.com/en-us/download/details.aspx?id=29884` (visited on 10/13/2022).

Different approaches cover the automatic identification of threats. For instance, Berger et al. [27] proposes an approach which extracts potential threats in a data flow diagram. Their threats are derived from CWEs and CAPECs. Briefly summarised, they analyse the data flow for possible dangerous patterns and mark these. In contrast, our approach does not search for these patterns but considers the propagation of attackers. A very similar to Berger et al. [27] data-flow-based threat detection approach is described in Tuma et al. [201], which tries to identify design flaws. Another data-flow-based approach is Sparta [179, 178].

Wortman et al. [220] provides an approach that generates an attack tree based on a software architecture. The attack tree is created by first selecting a target node as the root for the attack tree, similar as in C4.2. Afterwards, it then identifies all possible paths to the target. The path elements are then compared against a manually specified vulnerability file. This file then fills the attack tree. This attack tree is then automatically analysed for risk and potential costs. Our approach differs in regard that we additionally consider the access control properties for the attack propagation, and for our C4.2, we provide additional filtering options to identify relevant attack paths. In contrast, we do not provide information about the risk or the cost. However, to our understanding, our results could be used in combination with the described risk estimation approach. Other similar attack tree generation approaches are, for instance, Eckhart et al. [52] or Lemaire et al. [101]. The first is a threat analysis tool in the industrial automation domain, which can generate its own attack tree. The second generates attack trees based on a modelled CPS.

Besides the attack tree identification, there are also various approaches for generating attack paths or graphs [98, 13]. For instance, Arat et al. [15] propose an attack path detection for CPSs in the industrial domain. Similar to our approach, they first create an attack graph based on the modelled entities. Afterwards, they find attack paths between two selected nodes. This behaviour is quite comparable to our target node and the `StartElementFilter` in C4.2. Furthermore, they have filter operations to filter irrelevant attack paths. However, in detail, the approaches differ in two parts. First, in the attack path generation. Based on their focus on CPSs, they calculate the communication between different elements based on the closeness of two elements. In contrast, in our approach, this is a dedicated model element because, in our case, it is important that also not physically close elements can communicate. The second difference is the consideration of access control

properties. Our approach uses a fine-grained access control model, where their approach does not support access control so far.

Another similar attack propagation approach is Polatidis et al. [148, 147]. Their approach can identify attack paths between two elements. The approach uses information extracted from the CVE, CVSS and other vulnerability information for the attack propagation. Afterwards, they rank the found attack paths to find the most relevant attack paths. Their ranking is based on a recommender system. Our approach uses similar or the same concepts like CVE or CVSS for the attack propagation. However, in contrast, we additionally consider context-based access control properties for the propagation. Furthermore, our approaches differ in the selection of relevant attack paths. In our case, we use different filter operations to identify attack paths. In their case, they use a recommender system with an internal ranking system. Deloglos et al. [48] provide an attack propagation analysis for CPS. However, they do not consider access control properties in the propagation. Wang et al. [216] proposes an attack graph analysis approach for IoT. It uses similar to our approach CVEs to determine vulnerability properties. In contrast to our approach, they quantify risk based on the vulnerabilities used in an attack path. However, they do not consider attack propagations based solely on access control policies, like in our approach. In their approach, a CVE vulnerability is necessary for the attack propagation. Other automatic attack analyses are Aksu et al. [8], Yuan et al. [223] Ghosh et al. [62]. These approaches add support for considering privileges. Hence, they support access control properties. However, their access control model is very limited in contrast to our approach. For instance, it differs only between a user or admin in some cases. In our case, we use a fine-grained context-based access control system, which can provide more detailed feedback. Furthermore, our approach uses a detailed software architecture with deployment and services (for C4.1) for calculating the propagation. In contrast, their approaches use something more similar to network topologies. Another similar approach to the mentioned ones is Ibrahim et al. [76]. They focus on the automatic creation of attack graphs for microservices based on docker files. However, they also use a very limited access control model. Other approaches based on the network topology are Sheyner et al. [171], Phillips et al. [142], and Jajodia et al. [82]. The first, Sheyner et al. [171], describes an approach to extract attack graphs based on the network topology by using a model checker. They consider mitigation techniques, such as firewalls or intrusion detection systems, in their approach. In contrast, our approach uses a more fine-grained

access control model and uses the software architecture. Phillips et al. [142] describe an approach which can generate attack graphs with different privileges. However, they do not consider logical connections between services and only describe their concept theoretically. Jajodia et al. [82] describes an approach for identifying attack paths to a target network element. They automatically extract the network topology and vulnerabilities by using network vulnerability scanners. In contrast to our attack analysis, they only use physical network connections and cannot use logical connections for the propagation.

An attack propagation approach using the software architecture is the Cyber Security Modeling Language (CySeMoL) [182, 181]. It is developed for enterprise architectures and calculates an attack graph. The attack graph is calculated based on the likelihood of a successful attack on an architectural element. The likelihood is determined by measuring the time till a professional pen-tester compromised the architectural element. Therefore, it needs for each component type such measurements. Based on the likelihood and the attack graph, CySeMol can then calculate the overall cyber-security risk for the modelled system. In contrast, our approach does not need the likelihood but reuses the knowledge stored in vulnerability databases. Nevertheless, for new components, we also need security experts to identify new security vulnerabilities within a component. Furthermore, they state that they did not focus on confidentiality attacks. In contrast, our approach support a fine-grained access control system for modelling confidentiality.

The previously discussed work mostly used attack trees or directed attack graphs to express attacker and their behaviour. Chen et al. [39] presents an approach using Petri nets to model attacks in smart grids. They argue that Petri nets are more expressible than attack trees for describing attacks. However, in their eyes, the creation of large models is problematic. Therefore, they propose to individually model Petri nets by different stakeholders and then merge these models. For merging these Petri nets, they provide a modelling language for identifying matching parts. In contrast to our approach, they focussed on the modelling part and merging of Petri nets and not on the analysis of the propagation.

Kramer et al. [99] developed an attack analysis for PCM. In contrast to our approach, they focus on the attack propagation of a physical attacker. In other words, the attacker physically manipulates architectural elements to get access. For instance, the attacker picks a lock to get physical access to the

hardware in a protected room. By getting physical access, they can manipulate the hardware, and access or manipulate the deployed software components. They do not consider virtual access control or software vulnerabilities. In other words, they do not support the propagation of attackers based on virtual attacks. Nevertheless, in contrast, our approach only supports limited physical vulnerabilities (c.f. Section 7.4 — Software Vulnerabilities). Hence, it might be a good idea to couple these analyses somehow.

Another model-based security and attack analysis is UMLsec [87, 86]. Like the previously mentioned SecureUML [106], UMLsec extends UML for security properties. It provides various different analyses for analysing security and confidentiality properties, such as secure information flow, secure communication link or confidentiality analysis. In Ahmadian et al. [5], they presented their CARiSMA tool, which is based on the concepts of UMLSec and adds the support of context-based access control properties with the usage of Role-centric attribute-based access control [83]. In contrast, to their approach, our metamodel is more focused on the software architecture with components and services. Furthermore, both approaches support dedicated attacker models. However, in our approach, we focussed on the usage of CWEs and CVEs to describe vulnerabilities and specific propagation rules for them. These are commonly used concepts in the industry to specify vulnerabilities. Therefore, using these concepts for attack propagation in existing software might be easier.

Another possibility to analyse the system is by systematically testing the system for vulnerabilities, which can also be automated. For instance, Holm [72] proposes an approach to emulate a red team (an internal team trying to attack the system). Their approach systematically checks for vulnerabilities in the network and tries to compromise these. In contrast, our approach does not necessarily require a running system. Therefore, our approach can be used during downtime or with what-if cases. Another similar runtime-based approach is described by GhasemiGol et al. [61]. They suggest extending attack graphs with information based on intrusion detection systems and, therefore, providing a more accurate attack graph of the current network state. In contrast, their approach might provide more detailed information but requires a running system and a well-calibrated intrusion detection system.

8.2.4. Industrial Tools & Approaches

In the field of automatic attack propagation, various different industrial tools exist. Bloodhound[9][10] identifies attack paths based on the Active Directory (AD), which is a Microsoft service for managing a domain. Bloodhound uses credentials as well as vulnerabilities to create an attack graph for the identification of attack paths. In contrast to our approach, Bloodhound is currently limited to attacks and credentials for the AD. In our approach, we are not limited to this.

A similar tool is code_shield[11]. The tool focuses on identifying attack propagation in cloud providers, such as Amazon AWS. They focus on access control properties and finding the correct permission. Therefore, their benefit is very similar to our contributions C3 and C4 because they can identify too wide permissions or too few permissions. Therefore, they also target credential changes. In contrast, our approach does provide similar feedback but is more generalizable. Our approach is not limited to a specific cloud provider domain and can handle more general types of vulnerabilities. However, our modelling effort might be higher.

Another similar tool is CrowdStrike Falcon[12]. It markets itself as a threat intelligence tool and provides insights into current attacks worldwide and security vulnerabilities within a system. It can show vulnerabilities and their exploitation. However, to our understanding, it does not consider the access control properties for the attack propagation.

Regarding threat modelling, there exist various different tools. One widely known tool is the *Microsoft Threat Modelling Tool*[13]. It is used with the SDL[14] and supports STRIDE. Another tool is Threat Dragon[15] by OWASP.

[9] *BloodHound Enterprise*. URL: `https://bloodhoundenterprise.io/` (visited on 03/20/2023).
[10] *BloodHound: Six Degrees of Domain Admin*. URL: `https://bloodhound.readthedocs.io/en/latest/` (visited on 03/20/2023).
[11] CodeShield GmbH. *Codeshield*. URL: `https://codeshield.io/` (visited on 03/20/2023).
[12] *The CrowdStrike Falcon® Platform: One Platform, Complete Protection*. crowdstrike.com. URL: `https://www.crowdstrike.com/falcon-platform/` (visited on 03/20/2023).
[13] *Microsoft Threat Modeling Tool*. URL: `https://learn.microsoft.com/en-us/azure/security/develop/threat-modeling-tool` (visited on 06/05/2023).
[14] Microsoft. *Security Development Lifecycle*. URL: `https://www.microsoft.com/en-us/download/details.aspx?id=29884` (visited on 10/13/2022).
[15] *OWASP Threat Dragon*. URL: `https://owasp.org/www-project-threat-dragon/` (visited on 06/05/2023).

In contrast to our approach, both tools focus on identifying and mitigating threats. They do not cover the attack propagation with fine-grained access control. Nevertheless, their findings could be integrated as input for our vulnerability models.

8.3. Related Work Summary

To summarise the related work, there are different existing approaches for modelling and analysing access control. On the on hand many of these approaches, such as [88, 106, 9, 57], also support context-based access control for fine-grained access decisions. Some of these approaches, such as [168, 106], are also targeting design time and software architectures. On the other side, we have many approaches for modelling and analysing attacks, such as [162, 87]. Some of these approaches, such as [162], are still manual or only consider single threats and do not use vulnerability chaining or propagation to detect combined attack paths. However, there exist also various attack path estimation or propagation approaches such as [220, 8], which can provide vulnerability chaining. Most of these approaches only consider a network topology such as [171, 82] and cannot consider the software architecture. Using the software architecture can provide insights regarding the deployment and the affected services. Therefore, it is beneficial to consider the software architecture. However, the existing architectural attack path estimation approaches, such as CySeMoL [182, 181], do not consider fine-grained access control policies for the attack propagation. Other industrial approaches, such as Bloodhound[16][17] or code_shield[18] consider the fine-grained access control properties. However, they focus on their application domain, such as Microsoft Active Directory or Amazon AWS. To our knowledge, there exist no approach on the intersection of fine-grained access control policy analysis and attack propagation based on the software architecture level. Our approach considers fine-grained context-based access control policies and supports the identification of attack propagations based on the software architecture.

[16] *BloodHound Enterprise.* URL: https://bloodhoundenterprise.io/ (visited on 03/20/2023).

[17] *BloodHound: Six Degrees of Domain Admin.* URL: https://bloodhound.readthedocs.io/en/latest/ (visited on 03/20/2023).

[18] *The CrowdStrike Falcon® Platform: One Platform, Complete Protection.* crowdstrike.com. URL: https://www.crowdstrike.com/falcon-platform/ (visited on 03/20/2023).

It is not restricted to a certain application domain but can be used for any component-based software architecture.

9. Conclusion

This chapter concludes this thesis. We summarise our research and contributions in Section 9.1. There, we provide an overview of the key findings and outcomes of our work. Following the summary, we list the benefits of our approach for security experts and software architects in Section 9.2. We discuss how our access control metamodel, vulnerability metamodel and our analyses can lead to a more secure design. Finally, we conclude the thesis by outlining areas for future work in Section 9.3.

9.1. Summary

This thesis presented an approach for modelling and analysing access control policies and vulnerabilities within a software architecture. We developed four contributions and evaluated them on different evaluation scenarios. In the following, we will summarise the results for each contribution and then summarise the evaluation. The overall benefits of our approach are described in the next section (c.f. Section 9.2).

We start the summary by providing an overview of the modelling artefacts and their relationship to the security terms defined in Section 2.2.2. Figure 9.1 illustrates the security terms and how they relate to each other and our artefacts. The security terms are the boxes with solid borders. Boxes with dashed lines represent the artefacts. Our Common Vulnerabilities and Exposure (CVE) and Common Weakness Enumeration (CWE) vulnerabilities in our approach represent the concept of a *Vulnerability*. CVEs describes in our approach concrete vulnerabilities and CWEs describe classes of vulnerabilities. These vulnerabilities can be exploited by *Threats*. These threats are represented in our model by the potential attacks and the credentials of the attackers. If these threats are executed, they create compromised architectural elements or access violations. These are then the *Security Events* in

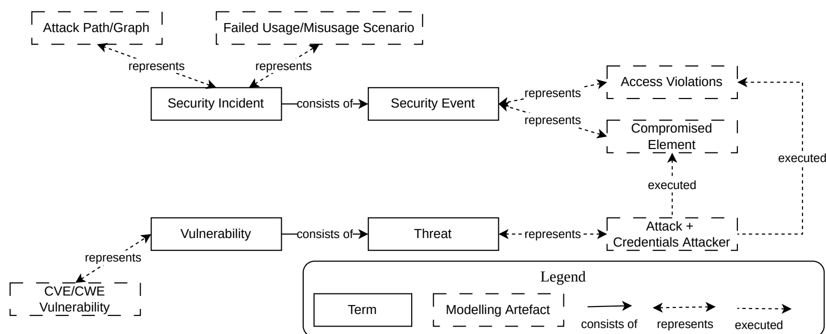


Figure 9.1.: Overview of the approach related to the security terms defined by ISO 27000 [77]

the terminology. Multiple of these potential events build than the *Security Incident*. A representation of the Security Incident is the attack graph/paths or the failed usage and misusage scenarios. It is important to mention that, in our case, the *Security Incident* and *Security Event* are both only a potential *Security Incident* and potential *Security Event* because they are based on our analysis and not real incidents.

Summarising our research questions, **RQ1** investigates how violations in software architecture can be identified. In more detail, we investigate in **RQ1.1** how access control properties are modelled in relation to the software architecture. These models are then the foundation for our research question **RQ1.2**, where we investigate, how we can analyse these access control policies. Our second research question **RQ2** investigate how attack propagation can be identified in relation to the software architecture. The research question **RQ2.1** investigates the relevant properties for an attack propagation and the research question **RQ2.2** investigates how we can analyse software architectures for attack propagations. In the following, we summarise our answers based on our contributions.

C1 Access Control Metamodel Our first contribution is the access control metamodel, which addresses our research question **RQ1.1**. In addition, it is used together with C2 to answer research question **RQ2.1**. The access control metamodel provides means to model context-based access control policies for software architectures. It is based on eXtensible Access Control Markup Language (XACML), an industry standard for modelling Attribute Based

Access Control (ABAC) policies. We provide custom model elements for the integration into PCM. This includes new selection elements (in XACML called match) to select PCM elements and assign them access control policies for protection. These access control policies rely on attributes for access decisions, and we also integrated these attributes within the software architecture. In addition, we introduce means to specify attribute assignments within usage scenarios. Furthermore, we add a new usage scenario type to PCM to model malicious user behaviours with misusage scenarios. In our overview in Figure 9.1, the *Credentials of the Attackers* and the *Failed Usage/Misusage Scenarios* are based on our contribution C1. Overall, our metamodel enables security experts to document access control properties within the software architecture.

Besides the metamodel, we developed a transformation from access control policies modelled with our access control metamodel to valid XACML files. Thereby enabling the usage of the modelled policies during system runtime.

C2 Vulnerability Metamodel Our second contribution is the metamodel for modelling vulnerabilities, attacks, and attackers within the software architecture. In conjunction with the previous contribution (C1), it answers our research question **RQ2.1**. Our vulnerability metamodel reuses and extends commonly used concepts in specifying and classifying software vulnerabilities, such as CVE or Common Vulnerability Scoring System (CVSS). These concepts are integrated into the software architecture and can be assigned to different architectural elements, such as components or processing units. Furthermore, we provide the means to model attack capabilities and knowledge with metamodel elements for attack and attackers. Regarding the attackers, we provide two different types of attackers. The first type uses concrete knowledge and capabilities to represent an attack propagation. The second type uses filters and knowledge to restrict the solution space for attacks. This contribution is the foundation for the *CVE/CWE Vulnerability*, the *Attack*, the *Compromised Element* and the *Attack Path/Graph* in our overview in Figure 9.1. This contribution enables security experts to model vulnerabilities within the software architecture.

C3 Scenario-Based Access Usage Analysis Our first analysis examines different usage and misusage scenarios regarding access violations. It answers our research question **RQ1.2**. The analysis checks each service call for violations

and then marks each usage or misusage scenario as either passed or not passed. We use an externally Policy Decision Point (PDP) for XACML to decide whether a service call is possible. A PDP evaluates access requests regarding the specified access control policies. The contribution C3 enables us to identify the *Failed Usage/Misusage Scenario*s in our overview Figure 9.1, representing potential security incidents. Hence, the benefit of this analysis is to identify potential security incidents before they occur.

C4.1 Attack Propagation Analysis Our second security analysis and first attack analysis is the attack propagation analysis. It helps to answer our research question **RQ2.2**. The analysis aims to provide a list of affected architectural elements. Affected architectural elements are elements compromised by an attacker. To achieve this goal, the analysis propagates from a starting point in the architecture and identifies all reachable architectural elements. This propagation uses vulnerabilities in architectural elements and access control policies an attacker can exploit with their knowledge. The knowledge and capabilities of attackers restrict the propagation. For instance, not all vulnerabilities can be exploited by all attackers. However, during the propagation, attackers can gain knowledge and use this new knowledge for further propagation steps. In the end, software architects get a list of affected components, services, data, hardware devices and network nodes. In our overview Figure 9.1, the potential *Security Incident* due to the *Attack Graph* are produced by this contribution. Hence, the benefit is that security experts become aware of these potential incidents and can discuss them with software architecture to mitigate them potentially.

C4.2 Targeted Attack Graph Analysis Our last contribution helps in answering the research question **RQ2.2**. In contrast to the attack propagation analysis, the target attack graph analysis calculates attack paths leading to a specific target within a software architecture. The detected potential attack paths also use vulnerabilities and access control properties. However, unlike the propagation analysis, which uses explicitly modelled attacks, C4.2 restricts attacks only based on their properties. Furthermore, the analysis considers different filter criteria, such as the starting point or the initially available credential, to calculate its attack paths. The results are attack paths leading to the targeted element. These attack paths are also the potential *Security Incident* in our overview Figure 9.1. This brings the security experts

the benefit of being aware of potential security incidents. Hence, they can, together with the software architect, try to mitigate the potential security incidents.

We evaluated our contributions C3, C4.1, C4.2 by creating for each contribution a GQM plan and using different evaluation scenarios. The two metamodel contributions (C1, C2) are not separately evaluated. However, they are indirectly evaluated based on their pragmatics because they are used in our analyses. Therefore, if we cannot trace issues in the analysis evaluation to the input types, we can assume that the metamodels are sufficient for our investigated cases.

During the evaluation of the contribution C3, we investigated four evaluation scenarios to assess the accuracy of our approach. In all scenarios, the analysis identifies all access decisions correctly, which indicates a high accuracy of our approach.

We evaluated the contribution C4.1 regarding the accuracy, effort reduction and scalability. To assess accuracy and effort reduction, we used 18 scenarios and sub-scenarios. The evaluation scenarios are based on real-world system breaches and commonly used evaluation cases. In the evaluation scenarios, we achieved a high accuracy which indicates a high overall accuracy. In addition, we achieved some effort reduction in most evaluation scenarios, and we discussed potential avenues for further improvements to reduce the effort. The scalability evaluation showed for smaller software architecture a reasonable runtime, but for larger systems (>=10,000 elements), the runtime increases significantly.

Similar to the attack propagation analysis, we evaluated C4.2 regarding the accuracy, effort reduction and scalability. For the accuracy evaluation, we investigated five evaluation scenarios with 52 possible attack paths. The evaluation scenarios are again based on real-world breaches and evaluation cases from related approaches. Although our overall accuracy results were not as high as those in the attack propagation analysis (C4.1), they were still satisfactory. This was due to the fact that our precision remained very high at 1.00, indicating that all identified attack paths are real attack paths, and we successfully identified attack paths for most elements (see Section 7.3.3). The main reason for the lower accuracy is the restriction to simple paths, and we believe that removing this restriction will yield better analysis results. However, this can come with performance drawbacks. Regarding the effort reduction, we illustrated some effort reduction compared to a manual analysis.

As for scalability, our results are better than those in the attack propagation analysis. Nevertheless, the runtime still increases significantly for larger architectures without filters.

9.2. Benefits

Based on our contributions, the application of our approach can bring various benefits to software architects or security experts. The main benefit is to increase the understanding of security properties for a software architecture and thereby increase the security of the overall architecture. In the following, we describe the benefits in more detail.

Documentation of Security Properties The first identified benefit is based on our contributions C1 and C2. The two metamodels are used to model explicitly the access control policies and vulnerabilities. In addition, they put them in relation to the software architecture. This enables software architects or security experts, for instance, to quickly see whether a component is vulnerable. In addition, it provides information about where certain access control properties can be gained in the system. These documented security properties can be a starting point for securing the system. Furthermore, the documentation gives security experts a common basis for discussing security properties with software architects. The models can also be used to discuss the security with external stakeholders like management or security consultants. We did not directly evaluated the quality of the metamodel for documentation. However, the evaluation of our analyses (C3 and C4) indicates the appropriateness of the metamodels through the used pragmatics (c.f. Section 2.1.1 and Chapter 7). Also, the generated attack graphs can be used for discussions with management because attack graphs may be better suited for illustrating cyberattacks than a list-based approach [144].

Better understanding of Access Control Policy Impact As previously described, understanding the impact of access control policies in complex systems can be challenging, especially for context-based access control policies like in our contribution C1. For instance, a single access request could depend on multiple different access control policies due to target selection in the

policies. Our contribution C3 tries to increase the understanding of the impact of access control policies by giving feedback on whether specific usage scenarios are possible. This feedback is given by detecting violations in usage and misusage scenarios. Our evaluation with our goal **G1** indicates that our approach can identify these violations in the given scenarios. Based on the high accuracy in the evaluation, we assume that the results also increase the understanding of access control policy impact. This benefit applies to multiple aspects. The software architects can check already during the design time whether the access control policies are sufficient to enable the intended usage. At the same time, they get feedback on whether the policies are restrictive enough to prevent malicious usage. Furthermore, they also can analyse the impact of policy changes in evolution steps. In addition, with our contribution C4, our approach provides information on whether attackers can abuse the access control policies to compromise the system.

Better understanding of Vulnerability Impact Our contribution C4 enables software architects or security experts to better understand the impact of vulnerabilities on the software architecture. As previously discussed, understanding the impact of vulnerabilities relies on various factors, such as the required privileges for exploitation and necessary attack vectors. Hence, for considering the impact, the exploitation context is essential. This is especially relevant for vulnerability chaining, where multiple vulnerabilities are combined. Here, our attack analyses can provide valuable feedback because they evaluate whether a vulnerability can be exploited and its potential effects on the system, as our analyses provide a list of compromised elements. Our evaluation for the attack analyses (C4) indicates that we can identify attack paths and attack graphs. These paths and graphs use vulnerabilities. Hence, they can provide an impact for vulnerabilities. Furthermore, through the graphs and paths, this information allows security experts to assess whether a vulnerability can be exploited and whether it leads to further compromisation. Then, they can discuss these with the software architects to identify mitigation techniques.

Identification of Mitigation Locations This benefit builds upon the previous one. In detail, it builds upon the list of affected architectural elements provided by our contribution C4. In our evaluation, we showed that we can generate this list with a high accuracy. The list of affected architectural elements

provides security experts with the insights into which vulnerabilities are used and how an attacker can move within a system. This knowledge is beneficial because it can be used to identify mitigation locations to break attack paths. For example, if all attack paths go over a specific component and exploit a vulnerability there, mitigating that vulnerability can effectively break the attack paths, thereby enhancing the system's security by reducing potential attack routes. The goal is to prevent, if possible, all attack propagations or at least ensure that attack paths only lead to non-critical architectural elements.

9.3. Future Work

In this section, we discuss possible future extensions to our approach. These possible extensions consist of scientific contributions, such as the analysis of further security properties, improving the evaluation, or new technical extensions, such as more architectural elements.

Supporting Mitigation Action The main future work for our approach is the consideration of mitigation approaches for the attack analyses. Currently, we consider only specific mitigation techniques, such as network segregation or access control. However, other techniques, such as encryption or the handling of compromised hardware elements, are so far not considered. To address this limitation, we could integrate a new metamodel element for mitigation. This metamodel element represents the mitigation techniques. This mitigation element has different subtypes for different mitigation techniques. For instance, trusted execution environments [158] can prevent the implicit takeover for components based on compromised `ResourceContainers`. Another mitigation type can be the mitigation of certain CWEs or CVEs. Here, annotating an architectural element with this mitigation would block the ability of an attacker to exploit the specified CWEs or CVEs. Besides mitigating a vulnerability, the different mitigation techniques are usually also vulnerable to certain attacks. In this regard, we plan to reuse our vulnerability model to specify possible vulnerabilities which can circumvent the mitigation. This extension could enable architects to analyse the effects of different mitigation techniques on the system and also proactively prepare for scenarios where mitigation measures could be compromised.

Considering User Actions in Attacks During an attack, attackers sometimes require the cooperation of other users in the system. Such cooperation can be involuntary or accidental, as seen in phishing attempts within an organization. In other scenarios, it can involve an internal accomplice intentionally assisting in the attack. Our vulnerability metamodel has an attribute based on the CVSS classification that indicates whether a vulnerability requires user interaction. We currently use it only within our attack path filters in C4.2. However, this usage can be extended by considering the usage scenarios of PCM. The usage scenarios describe the grouped user behaviour. To realize this feature, we need a mapping between user action and vulnerability. A solution could be that we reference prerequisite actions for exploiting in our vulnerability metaclass. These prerequisite actions can be system services. In addition, we need to extend the attacker model to select the usage behaviour. With these extensions, we can then check whether there exists a service call in the referenced usage scenarios during the exploitation. This service call would then represent the user interaction.

Considering Additional Security Impacts Our vulnerability metamodel contains more impact information from CVSS regarding a vulnerability as we use in our attack analyses. Currently, we only use the *confidentiality* impact. However, CVSS also provides impact descriptions for *integrity* and *availability*. Integrity refers to an attacker's ability to manipulate or change services, while availability describes an attacker's potential to take down a service, therefore, making it unavailable. As a potential integration within our attack analysis, we could combine it with the possible extensions regarding attacker states and provide a list of affected architectural elements for each impact. For instance, for availability, the output could be a list of architectural elements where attackers can affect the availability. The same can be done for integrity. This can even be extended to not yet considered impact descriptions.

More Detailed Consideration of Data Our attack analyses do not consider data flow properties. However, incorporating data flow analysis results could improve the data extraction process for determining the affected data. For instance, for compromised `LinkingResources`, the analysis could identify unencrypted data flowing through them. To achieve this example, the analysis is required to reuse the results of a data flow analysis to identify the data and identify the characteristics of the data. The characteristics are metadata,

such as encrypted or unencrypted. In Walter et al. [209], we developed an initial prototype for such a coupling. We used the data flow analysis for PCM [168] to estimate the criticality of data elements and combined it without our attack propagations. This combination enabled us to assess the overall criticality of attack propagations.

Supporting Self-Adapting Access Control Approaches As previously described in our limitations (c.f. Section 7.4 — Static Access Control Policies), our access control policies do not support self-adaptation. While adding support for self-adaptation could also be future work, these approaches are usually more runtime-orientated. There are approaches such as Simulizar [24] for performance simulation, which could be utilised to enable self-adaptation for design time approaches. However, another interesting area is to leverage our developed attack analyses to provide insights into the impact of access decisions and use this knowledge for the self-adaptation process at runtime. In Walter et al. [209], we already provided similar feedback to a non-self-adapting runtime access control system. In that work, we analysed the critical data that an attacker can access if access to a specific architectural element is granted. Therefore, this approach is similar to risk-based access control approaches (c.f. Section 8.1.1). We calculate the criticality of the data by first identifying the affected data. For identifying the data, we utilise our contribution C4.1 and select as a start point the architectural element to which the access is granted. A similar approach could be realised with Bureš et al. [34, 35], where our approach could provide the affected elements as input for the self-adaption. This integration would support self-adapting access control approaches by gaining insights into the impact of an access decision.

Increased Support for Automatic Model Generation One important area of future work involves improving the automatic model generation process. We presented the first prototype for automatically deriving security properties in Kirschner et al. [94] and Section 4.3.5. Such approaches have the potential to significantly reduce the modelling effort, making them particularly valuable for legacy systems. However, the current state of the approach still has some drawbacks regarding the architectural recovery and the vulnerability derivation from source code. With respect to the architectural recovery, the approach cannot recover all necessary architectural elements, such as the `AssemblyContexts` or the `ResourceContainers`. Regarding the vulnerability

derivation, the approach is limited to the concrete CVEs the used source analysis can identify. However, newer versions of the used source-code analysis can also identify potential CWEs in the source code. Here, an open point would be how to handle the other missing vulnerability classification properties. For the CVEs, we can automatically derive most of the properties based on their CVE classification. However, for the CWEs, it does not yet exist. A possible solution could be that this additional information needs to be manually specified.

Improved Approach Evaluation One open topic for our analysis is the further evaluation of our approach. This covers multiple topics. First, the evaluation of the metamodels. Currently, we only evaluate the metamodel by the application in our security analyses. While this is sufficient for our intended usage, evaluating the model itself can be potentially beneficial. This can also be relevant if the metamodel is used for the documentation. Here, especially the usability is important. This can be evaluated by performing a user study where participants should model different scenarios. Another aspect for the metamodel and the analysis are the applicability or expressiveness. In this regard, it is for security experts important whether the vulnerability metamodel can be used to model commonly used vulnerabilities or attackers. Other aspects can be whether the access control policy metamodel can be used to model commonly used access control policies. This evaluation could also provide information regarding whether we considered really all important security properties and whether we considered all propagation rules.

Better Scalability for Attack Analyses During our evaluation, we identified runtime scalability problems with our different attack analyses. In the future, it could be beneficial to analyse the concrete runtime problems and improve the scalability. The first starting point for this problem is the parallelisation of the attack propagation analysis. In our current state, the attack propagation analysis is a single-thread application. However, the attack propagation rules are mostly independent of each other. Therefore, they can be executed in parallel. In addition, selecting only relevant attack propagation rules based on the previous propagation is possible, thereby reducing the necessary number of checks. In addition, it may be good to change the underlying data structure to a more efficient one. This change in the data structure is also beneficial for the second attack analysis. For the second analysis, the chosen graph

framework could also be replaced with a more efficient one. This modification could enable faster results or even bring better results if we can relax our assumption about simple paths.

Support for more Architectural Elements Currently, our approach only considers a subset of the available architectural elements within PCM. For the future, it can be beneficial to extend the support. Especially the support of `CompositeComponents` can be useful. This type can help to specify components with subcomponents and enables a more fine-grained specification of vulnerabilities. The challenging factor here is that instantiated `CompositeComponents` are not expressed directly in PCM but indirectly derived. Especially for handling multiple subcomponents which build a hierarchy, the handling is slightly different from other PCM elements. Our contributions C1, C2 C3 already support the specification and analysis. We realized this integration similar to other PCM analyses with modelling lists of `AssemblyContexts`. However, the attack analyses C4 are missing the support. There the propagation rules and the output needs to be updated. Besides the support of `CompositeComponents`, other PCM elements, such as the interfaces for infrastructure calls, could be beneficial. These additional elements would enable a more fine-grained assignment of vulnerabilities. In our eyes, the integration should be possible. It would require adding the new architectural elements to the metamodel and then, for the attack analyses, the specification of new propagation rules. For the scenario analysis, it would require a slight adjustment on the SEFF finder.

So overall, in this thesis, we investigated the impact of access control policies and vulnerabilities for the security of a system. We developed two metamodels to express the software architecture's access control policies and vulnerabilities. These metamodels are used in our three security analyses to estimate the impact of access violations and attack paths. There are still open questions regarding the impact of mitigations or the impact of other security properties such as integrity or availability. However, our developed approach is a foundation to further investigate these questions in the future.

Acknowledgement

During the writing of the thesis, I used the following tools to improve the writing and the grammar:

- Grammarly[1]
- LanguageTool[2]
- ChatGPT[3]
- Google Bard[4]

However, the ideas and argumentation are based on my own ideas.

1 https://www.grammarly.com/
2 https://languagetool.org/de
3 https://openai.com/blog/chatgpt
4 https://bard.google.com/

Part V.

Appendix

Bibliography

[1] *A04 Insecure Design - OWASP Top 10:2021*. URL: https://owasp.org/Top10/A04_2021-Insecure_Design/ (visited on 04/03/2023).

[2] *About the Meta Object Facility Specification Version 2.5.1*. URL: https://www.omg.org/spec/MOF (visited on 04/12/2023).

[3] *About the Unified Modeling Language Specification Version 2.5*. URL: https://www.omg.org/spec/UML/2.5 (visited on 04/12/2023).

[4] ACM. *Artifact Review and Badging Version 1.1*. Aug. 24, 2020. URL: https://www.acm.org/publications/policies/artifact-review-and-badging-current (visited on 02/27/2023).

[5] A. S. Ahmadian, S. Peldszus, Q. Ramadan, and J. Jürjens. "Model-based privacy and security analysis with CARiSMA". In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2017. ACM, Aug. 21, 2017, pp. 989–993. ISBN: 978-1-4503-5105-8. DOI: 10.1145/3106237.3122823.

[6] A. V. Aho and J. D. Ullman. *The Theory of Parsing, Translation, and Compiling*. USA: Prentice-Hall, Inc., 1972. ISBN: 0139145567. URL: https://dl.acm.org/doi/book/10.5555/578789 (visited on 01/09/2023).

[7] M. Ait El Hadj, M. Ayache, Y. Benkaouz, A. Khoumsi, and M. Erradi. "Clustering-based Approach for Anomaly Detection in XACML Policies:" in: *Proceedings of the 14th International Joint Conference on e-Business and Telecommunications*. 14th International Conference on Security and Cryptography. SCITEPRESS - Science and Technology Publications, 2017, pp. 548–553. ISBN: 978-989-758-259-2. DOI: 10.5220/0006471205480553.

[8] M. U. Aksu, K. Bicakci, M. H. Dilek, A. M. Ozbayoglu, and E. I. Tatli. "Automated Generation of Attack Graphs Using NVD". In: *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy (ODASPY)*. ACM, 2018, pp. 135–142. ISBN: 9781450356329. DOI: 10.1145/3176258.3176339.

[9] F. Alberti, A. Armando, and S. Ranise. "Efficient symbolic automated analysis of administrative attribute-based RBAC-policies". In: *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security - ASIACCS 11*. ACM, 2011, p. 165. ISBN: 978-1-4503-0564-8. DOI: 10.1145/1966913.1966935.

[10] N. Alhebaishi, L. Wang, S. Jajodia, and A. Singhal. "Threat modeling for cloud data center infrastructures". In: *Foundations and Practice of Security - 9th International Symposium, FPS*. Springer, 2016, pp. 302–319. ISBN: 978-3-319-51966-1. DOI: 10.1007/978-3-319-51966-1_20.

[11] R. Al-Ali, T. Bures, B.-O. Hartmann, J. Havlik, R. Heinrich, P. Hnetynka, A. Juan-Verdejo, P. Parizek, S. Seifermann, and M. Walter. *Use Cases in Dataflow-Based Privacy and Trust Modeling and Analysis in Industry 4.0 Systems*. Karlsruhe Reports in Informatics 9. Karlsruher Institut für Technologie (KIT), 2018. 43 pp. DOI: 10.5445/IR/1000085169.

[12] R. Al-Ali, R. Heinrich, P. Hnetynka, A. Juan-Verdejo, S. Seifermann, and M. Walter. "Modeling of dynamic trust contracts for industry 4.0 systems". In: *Proceedings of the 12th European Conference on Software Architecture: Companion Proceedings*. 12th European Conference on Software Architecture. ECSA 2018. ACM, 2018, 45:1–45:4. ISBN: 978-1-4503-6483-6. DOI: 10.1145/3241403.3241450.

[13] O. S. M. B. H. Almazrouei, P. Magalingam, M. K. Hasan, and M. Shanmugam. "A Review on Attack Graph Analysis for IoT Vulnerability Assessment: Challenges, Open Issues, and Future Directions". In: *IEEE Access* 11 (2023), pp. 44350–44376. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2023.3272053.

[14] R. Alur and D. Dill. "The theory of timed automata". In: *Real-Time: Theory in Practice*. Ed. by J. W. de Bakker, C. Huizing, W. P. de Roever, and G. Rozenberg. Berlin, Heidelberg: Springer Berlin Heidelberg, 1992, pp. 45–73. ISBN: 978-3-540-47218-6. DOI: 10.1007/BFb0031987.

[15] F. Arat and S. Akleylek. "Attack Path Detection for IIoT Enabled Cyber Physical Systems: Revisited". In: *Computers & Security* 128 (2023), p. 103174. ISSN: 0167-4048. DOI: 10.1016/j.cose.2023.103174.

[16] H. F. Atlam, M. A. Azad, M. O. Alassafi, A. A. Alshdadi, and A. Alenezi. "Risk-Based Access Control Model: A Systematic Literature Review". In: *Future Internet* 12.6 (June 2020), p. 103. ISSN: 1999-5903. DOI: 10.3390/fi12060103.

[17] M. El-Attar. “From misuse cases to mal-activity diagrams: bridging the gap between functional security analysis and design”. In: *Software & Systems Modeling* 13.1 (Feb. 2014), pp. 173–190. ISSN: 1619-1366, 1619-1374. DOI: 10.1007/s10270-012-0240-5.

[18] M. S. Barik, A. Sengupta, and C. Mazumdar. “Attack Graph Generation and Analysis Techniques”. In: *Defence Science Journal* 66.6 (Oct. 31, 2016), p. 559. ISSN: 0976464X, 0011748X. DOI: 10.14429/dsj.66.10795.

[19] G. Basili, V. R. Caldiera, and H. D. Rombach. “The goal question metric approach”. In: *Encyclopedia of software engineering* (1994), pp. 528–532.

[20] V. R. Basili and D. M. Weiss. “A Methodology for Collecting Valid Software Engineering Data”. In: *IEEE Transactions on Software Engineering* SE-10.6 (1984), pp. 728–738. DOI: 10.1109/TSE.1984.5010301.

[21] D. Basin, M. Clavel, J. Doser, and M. Egea. “Automated analysis of security-design models”. In: *Information and Software Technology* 51.5 (May 1, 2009), pp. 815–831. ISSN: 0950-5849. DOI: 10.1016/j.infsof.2008.05.011.

[22] F. L. Bauer. “Encryption”. In: *Encyclopedia of Cryptography and Security*. Ed. by H. C. A. van Tilborg. Boston, MA: Springer US, 2005, pp. 202–202. ISBN: 978-0-387-23483-0. DOI: 10.1007/0-387-23483-7_141.

[23] L. Bauer, S. Garriss, and M. K. Reiter. “Detecting and resolving policy misconfigurations in access-control systems”. en. In: *ACM Transactions on Information and System Security* 14.1 (May 2011), pp. 1–28. ISSN: 1094-9224, 1557-7406. DOI: 10.1145/1952982.1952984.

[24] M. Becker, S. Becker, and J. Meyer. “Simulizar: Design-time modeling and performance analysis of self-adaptive systems”. In: *Software Engineering 2013* (2013).

[25] S. Becker, M. Hauck, M. Trifu, K. Krogmann, and J. Kofroň. “Reverse Engineering Component Models for Quality Predictions”. In: *European Conference on Software Maintenance and Reengineering*. IEEE, 2010, pp. 194–197. DOI: 10.1109/CSMR.2010.34.

[26] S. Becker, S. Dziwok, C. Gerking, C. Heinzemann, W. Schäfer, M. Meyer, and U. Pohlmann. "The MechatronicUML Method: Model-Driven Software Engineering of Self-Adaptive Mechatronic Systems". In: *Companion Proceedings of the 36th International Conference on Software Engineering*. ICSE Companion 2014. Hyderabad, India: ACM, 2014, pp. 614–615. ISBN: 9781450327688. DOI: 10.1145/2591062.2591142.

[27] B. J. Berger, K. Sohr, and R. Koschke. "Automatically extracting threats from extended data flow diagrams". In: *Engineering Secure Software and Systems - 8th International Symposium (ESSoS)*. Lecture Notes in Computer Science. Springer, 2016, pp. 56–71. DOI: 10.1007/978-3-319-30806-7_4.

[28] E. Bertino, P. A. Bonatti, and E. Ferrari. "TRBAC: A temporal role-based access control model". In: *ACM Transactions on Information and System Security* 4.3 (Aug. 1, 2001), pp. 191–233. ISSN: 1094-9224. DOI: 10.1145/501978.501979.

[29] E. Bertino, A. A. Jabal, S. Calo, D. Verma, and C. Williams. "The Challenge of Access Control Policies Quality". In: *Journal of Data and Information Quality* 10.2 (Sept. 7, 2018), pp. 1–6. ISSN: 19361955. DOI: 10.1145/3209668.

[30] A. Bertolino, S. Daoudagh, F. Lonetti, and E. Marchetti. "An Automated Testing Framework of Model-Driven Tools for XACML Policy Specification". In: *9th International Conference on the Quality of Information and Communications Technology*. Sept. 2014, pp. 75–84. DOI: 10.1109/QUATIC.2014.17.

[31] A. Bertolino, S. Daoudagh, F. Lonetti, and E. Marchetti. "Automatic XACML Requests Generation for Policy Testing". In: *Verification and Validation 2012 IEEE Fifth International Conference on Software Testing*. ISSN: 2159-4848. Apr. 2012, pp. 842–849. DOI: 10.1109/ICST.2012.185.

[32] N. Boltz, S. Hahner, M. Walter, S. Seifferman, R. Heinrich, T. Bureš, and P. Hnětynka. "Handling Environmental Uncertainty in Design Time Access Control Analysis". In: *48th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE, 2022, pp. 382–389. DOI: 10.1109/SEAA56994.2022.00067.

[33] N. Boltz, M. Walter, and R. Heinrich. "Context-Based Confidentiality Analysis for Industrial IoT". In: *46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE, Aug. 2020, pp. 589–596. ISBN: 978-1-72819-532-2. DOI: 10.1109/SEAA51224.2020.00096.

[34] T. Bureš, P. Hnětynka, M. Kruliš, F. Plášil, D. Khalyeyev, S. Hahner, S. Seifermann, M. Walter, and R. Heinrich. "Attuning Adaptation Rules via a Rule-Specific Neural Network". In: *Leveraging Applications of Formal Methods, Verification and Validation. Adaptation and Learning (ISOLA)*. Ed. by T. Margaria and B. Steffen. Springer, 2022, pp. 215–230. ISBN: 978-3-031-19759-8. DOI: 10.1007/978-3-031-19759-8_14.

[35] T. Bureš, P. Hnětynka, M. Kruliš, F. Plášil, D. Khalyeyev, S. Hahner, S. Seifermann, M. Walter, and R. Heinrich. "Generating Adaptation Rule-Specific Neural Network". In: *International Journal on Software Tools for Technology Transfer* (). accepted, to appear.

[36] K. Busch. "An Architecture-based Approach for Change Impact Analysis of Software-intensive Systems". PhD thesis. Karlsruher Institut für Technologie (KIT), 2019. 275 pp. DOI: 10.5445/IR/1000097837.

[37] M. Busch. "Evaluating & Engineering: an Approach for the Development of Secure Web Applications". PhD thesis. Muinch, Germany: Ludwig-Maximilians-Universität München, 2016. 215 pp. URL: https://www.pst.ifi.lmu.de/~busch/thesisMarianneBusch.pdf (visited on 03/22/2023).

[38] *CAPEC - Common Attack Pattern Enumeration and Classification (CAPEC™)*. URL: https://capec.mitre.org/ (visited on 10/25/2021).

[39] T. M. Chen, J. C. Sanchez-Aarnoutse, and J. Buford. "Petri Net Modeling of Cyber-Physical Attacks on Smart Grid". In: *IEEE Transactions on Smart Grid* 2.4 (Dec. 2011), pp. 741–749. ISSN: 1949-3053. DOI: 10.1109/TSG.2011.2160000.

[40] E. Cole. *Advanced persistent threat: understanding the danger and how to protect your organization*. Newnes, 2012. ISBN: 978-1597499491.

[41] M. Coscia. "Multidimensional network analysis". PhD thesis. Universitá Degli Studi Di Pisa, 2012.

[42] J. Crampton and C. Morisset. "PTaCL: A Language for Attribute-Based Access Control in Open Systems". In: *Principles of Security and Trust: First International Conference, POST 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS.* Ed. by P. Degano and J. D. Guttman. Springer, 2012, pp. 390–409. ISBN: 978-3-642-28641-4. DOI: 10.1007/978-3-642-28641-4_21.

[43] F. Cuppens and A. Miège. "Modelling contexts in the Or-BAC model". In: *19th Annual Computer Security Applications Conference (ACSAC).* IEEE, 2003, pp. 416–425. DOI: 10.1109/CSAC.2003.1254346.

[44] CVSS Special Interest Group (SIG). *CVSS 3.1.* URL: https://www.first.org/cvss/v3-1/cvss-v31-specification_r1.pdf (visited on 10/25/2021).

[45] CVSS Special Interest Group (SIG). *CVSS SIG.* URL: https://www.first.org/cvss/ (visited on 10/25/2021).

[46] CVSS Special Interest Group (SIG). *Vulnerability Chaining.* URL: https://www.first.org/cvss/v3.1/user-guide#3-4-Vulnerability-Chaining (visited on 03/14/2022).

[47] *Cyber Security Breaches Survey 2022.* en. Mar. 2022. URL: https://www.gov.uk/government/statistics/cyber-security-breaches-survey-2022/cyber-security-breaches-survey-2022 (visited on 04/03/2023).

[48] C. Deloglos, C. Elks, and A. Tantawy. "An Attacker Modeling Framework for the Assessment of Cyber-Physical Systems Security". In: *Computer Safety, Reliability, and Security - 39th International Conference, SAFECOMP.* Lecture Notes in Computer Science. Springer, 2020, pp. 150–163. ISBN: 978-3-030-54549-9. DOI: 10.1007/978-3-030-54549-9_10.

[49] A. van Den Berghe, R. Scandariato, K. Yskout, and W. Joosen. "Design notations for secure software: a systematic literature review". In: *Software & Systems Modeling* 16.3 (2017), pp. 809–831. DOI: 10.1007/s10270-015-0486-9.

[50] D. E. Denning. "A Lattice Model of Secure Information Flow". In: *Communication ACM* 19.5 (May 1976), pp. 236–243. ISSN: 0001-0782. DOI: 10.1145/360051.360056.

[51] C. Dwork. “Differential Privacy”. In: *Automata, Languages and Programming*. Ed. by M. Bugliesi, B. Preneel, V. Sassone, and I. Wegener. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2006, pp. 1–12. ISBN: 978-3-540-35908-1. DOI: 10.1007/11787006_1.

[52] M. Eckhart, K. Meixner, D. Winkler, and A. Ekelhart. “Securing the testing process for industrial automation software”. In: *Computers & Security* 85 (Aug. 2019), pp. 156–180. ISSN: 01674048. DOI: 10.1016/j.cose.2019.04.016.

[53] *Exploit Prediction Scoring System (EPSS)*. FIRST — Forum of Incident Response and Security Teams. URL: https://www.first.org/epss (visited on 03/28/2023).

[54] *FALLOUT: THE REPUTATIONAL IMPACT OF IT RISK*. Forbes Insights, 2014, p. 21. URL: https://images.forbes.com/forbesinsights/StudyPDFs/IBM_Reputational_IT_Risk_REPORT.pdf (visited on 08/30/2023).

[55] D. Ferraiolo, J. Cugini, and D. R. Kuhn. “Role-based access control (RBAC): Features and motivations”. In: *Proceedings of 11th annual computer security application conference*. 1995, pp. 241–248.

[56] A. Feutrill, D. Ranathunga, Y. Yarom, and M. Roughan. “The effect of common vulnerability scoring system metrics on vulnerability exploit delay”. In: *2018 Sixth International Symposium on Computing and Networking (CANDAR)*. IEEE, 2018, pp. 1–10. DOI: 10.1109/CANDAR.2018.00009.

[57] K. Fisler, S. Krishnamurthi, L. A. Meyerovich, and M. C. Tschantz. “Verification and change-impact analysis of access-control policies”. In: *27th international conference on Software engineering (ICSE)*. ACM, 2005, p. 196. DOI: 10.1145/1062455.1062502.

[58] M. Fujita, P. McGeer, and J.-Y. Yang. “Multi-Terminal Binary Decision Diagrams: An Efficient Data Structure for Matrix Representation”. In: *Formal Methods in System Design* 10.2 (Apr. 1997), pp. 149–169. ISSN: 1572-8102. DOI: 10.1023/A:1008647823331.

[59] S. Garg, R. Singh, and A. Mohapatra. “Analysis of software vulnerability classification based on different technical parameters”. In: *Information Security Journal: A Global Perspective* 28.1 (Mar. 4, 2019), pp. 1–19. ISSN: 1939-3555. DOI: 10.1080/19393555.2019.1628325.

[60] C. Gerking and D. Schubert. “Component-Based Refinement and Verification of Information-Flow Security Policies for Cyber-Physical Microservice Architectures”. In: *2019 IEEE International Conference on Software Architecture (ICSA)*. IEEE, Mar. 2019, pp. 61–70. ISBN: 978-1-7281-0528-4. DOI: 10.1109/ICSA.2019.00015.

[61] M. GhasemiGol, A. Ghaemi-Bafghi, and H. Takabi. “A comprehensive approach for network attack forecasting”. In: *Computers & Security* 58 (2016), pp. 83–105. ISSN: 0167-4048. DOI: 10.1016/j.cose.2015.11.005.

[62] N. Ghosh and S. K. Ghosh. “A planner-based approach to generate and analyze minimal attack graph”. In: *Applied Intelligence* 36.2 (Mar. 1, 2012), pp. 369–390. ISSN: 1573-7497. DOI: 10.1007/s10489-010-0266-8.

[63] D. P. Guelev, M. Ryan, and P. Y. Schobbens. “Model-Checking Access Control Policies”. In: *Information Security*. Ed. by K. Zhang and Y. Zheng. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2004, pp. 219–230. ISBN: 978-3-540-30144-8. DOI: 10.1007/978-3-540-30144-8_19.

[64] M. Hafner, R. Breu, B. Agreiter, and A. Nowak. “Sectet: an extensible framework for the realization of secure inter-organizational workflows”. In: *Internet Research* 16.5 (2006), pp. 491–506. ISSN: 1066-2243. DOI: 10.1108/10662240610710978.

[65] S. Hahner, T. Bitschi, M. Walter, T. Bureš, H. Petr, and R. Heinrich. “Model-based Confidentiality Analysis under Uncertainty”. In: *2023 IEEE 20th International Conference on Software Architecture Companion (ICSA-C)*. MDE4SA – 3rd International Workshop On Model-driven Engineering for Software Architecture. IEEE, 2023, pp. 256–263. DOI: 10.1109/ICSA-C57050.2023.00062.

[66] S. Hahner, R. Heinrich, and R. Reussner. “Architecture-based Uncertainty Impact Analysis to ensure Confidentiality”. In: *SEAMS*. IEEE/ACM, 2023. DOI: 10.1109/SEAMS59076.2023.00026.

[67] S. Hahner, S. Seifermann, R. Heinrich, M. Walter, T. Bureš, and P. Hnětynka. “Modeling Data Flow Constraints for Design-Time Confidentiality Analyses”. In: *2021 IEEE 18th International Conference on Software Architecture Companion (ICSA-C)*. IEEE, Mar. 2021, pp. 15–21. DOI: 10.1109/ICSA-C52384.2021.00009.

[68] B. A. Hamilton. *Industrial Cybersecurity Threat Briefing*. Tech. rep., p. 82.

[69] A. Hariri, S. Bandopadhyay, A. Rizos, T. Dimitrakos, B. Crispo, and M. Rajarajan. "SIUV: A Smart Car Identity Management and Usage Control System Based on Verifiable Credentials". In: *ICT Systems Security and Privacy Protection - 36th IFIP TC 11 International Conference, SEC*. Ed. by A. Jøsang, L. Futcher, and J. Hagen. IFIP Advances in Information and Communication Technology. Cham: Springer, 2021, pp. 36–50. ISBN: 978-3-030-78120-0. DOI: 10.1007/978-3-030-78120-0_3.

[70] R. Heinrich. "Architectural runtime models for integrating runtime observations and component-based models". In: *Journal of Systems and Software* 169 (Nov. 2020), p. 110722. ISSN: 01641212. DOI: 10.1016/j.jss.2020.110722.

[71] R. Heinrich, S. Koch, S. Cha, K. Busch, R. Reussner, and B. Vogel-Heuser. "Architecture-based change impact analysis in cross-disciplinary automated production systems". In: *Journal of Systems and Software* 146 (2018), pp. 167–185. ISSN: 0164-1212. DOI: 10.1016/j.jss.2018.08.058.

[72] H. Holm. "Lore a Red Team Emulation Tool". In: *IEEE Transactions on Dependable and Secure Computing* 20.2 (Mar. 2023), pp. 1596–1608. ISSN: 1941-0018. DOI: 10.1109/TDSC.2022.3160792.

[73] V. Hu et al. "Attribute-Based Access Control". In: *Computer* 48.2 (Feb. 2015), pp. 85–88. ISSN: 0018-9162. DOI: 10.1109/MC.2015.33.

[74] V. C. Hu, D. Ferraiolo, R. Kuhn, A. Schnitzer, K. Sandlin, R. Miller, and K. Scarfone. *Guide to Attribute Based Access Control (ABAC) Definition and Considerations*. NIST SP 800-162. National Institute of Standards and Technology, Jan. 2014, NIST SP 800–162. DOI: 10.6028/NIST.SP.800-162.

[75] G. Hughes and T. Bultan. "Automated verification of access control policies using a SAT solver". In: *International Journal on Software Tools for Technology Transfer* 10.6 (Dec. 1, 2008), pp. 503–520. ISSN: 1433-2787. DOI: 10.1007/s10009-008-0087-9.

[76] A. Ibrahim, S. Bozhinoski, and A. Pretschner. "Attack graph generation for microservice architecture". In: *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*. SAC '19. ACM, Apr. 8, 2019, pp. 1235–1242. ISBN: 978-1-4503-5933-7. DOI: 10.1145/3297280.3297401.

[77] ISO Central Secretary. *Information technology — Security techniques — Information security management systems — Overview and vocabulary*. en. Standard ISO/IEC 27000:2018. Geneva, CH: International Organization for Standardization, 2018. URL: https://www.iso.org/standard/73906.html (visited on 07/20/2023).

[78] ISO Central Secretary. *Information technology — The JSONdata interchange syntax*. Standard ISO 21778:2017(E). Geneva, CH: International Organization for Standardization, Nov. 2017. URL: https://www.iso.org/standard/71616.html (visited on 07/20/2023).

[79] A. A. Jabal, M. Davari, E. Bertino, C. Makaya, S. Calo, D. Verma, and C. Williams. "ProFact: A Provenance-based Analytics Framework for Access Control Policies". In: *Transactions on Services Computing* 14 (6 2019), pp. 1–1. ISSN: 1939-1374. DOI: 10.1109/TSC.2019.2900641.

[80] A. A. Jabal, M. Davari, E. Bertino, C. Makaya, S. Calo, D. Verma, A. Russo, and C. Williams. "Methods and Tools for Policy Analysis". In: *ACM Computing Surveys* 51.6 (Feb. 2019), pp. 1–35. ISSN: 03600300. DOI: 10.1145/3295749.

[81] J. Jacobs, S. Romanosky, B. Edwards, I. Adjerid, and M. Roytman. "Exploit Prediction Scoring System (EPSS)". In: *Digital Threats: Research and Practice* 2.3 (July 9, 2021), 20:1–20:17. ISSN: 2692-1626. DOI: 10.1145/3436242.

[82] S. Jajodia, S. Noel, and B. O'Berry. "Topological Analysis of Network Attack Vulnerability". In: *Managing Cyber Threats: Issues, Approaches, and Challenges* 5 (2005), pp. 247–266. DOI: 10.1007/0-387-24230-9_9.

[83] X. Jin, R. Sandhu, and R. Krishnan. "RABAC: Role-Centric Attribute-Based Access Control". In: *Computer Network Security - 6th International Conference on Mathematical Methods, Models and Architectures for Computer Network Security, MMM-ACNS*. Ed. by I. Kotenko and V. Skormin. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2012, pp. 84–96. ISBN: 978-3-642-33704-8. DOI: 10.1007/978-3-642-33704-8_8.

[84] E. Johns. *Cyber Security Breaches Survey 2021: Statistical Release*. en. Tech. rep. London, United Kingdom: Department for Digial, Culture, Media & Sport (DCMS), 2021, p. 66. URL: https://assets.publishing.service.gov.uk/government/uploads/system/uploads/attachment_data/file/972399/Cyber_Security_Breaches_Survey_2021_Statistical_Release.pdf (visited on 03/04/2023).

[85] P. Johnson, R. Lagerström, and M. Ekstedt. "A Meta Language for Threat Modeling and Attack Simulations". In: *Proceedings of the 13th International Conference on Availability, Reliability and Security (ARES).* ARES 2018. ACM, Aug. 27, 2018, pp. 1–8. ISBN: 978-1-4503-6448-5. DOI: 10.1145/3230833.3232799.

[86] J. Jürjens. *Secure Systems Development with UML.* Springer, 2004. ISBN: 978-3-540-00701-2. DOI: 10.1007/b137706.

[87] J. Jürjens. *UMLsec: Extending UML for Secure Systems Development.* Vol. 2460. Springer Berlin Heidelberg, 2002, pp. 412–425. ISBN: 978-3-540-44254-7. DOI: 10.1007/3-540-45800-X_32.

[88] A. A. E. Kalam, R. E. Baida, P. Balbiani, S. Benferhat, F. Cuppens, Y. Deswarte, A. Miege, C. Saurel, and G. Trouessin. "Organization based access control". In: *Proceedings POLICY 2003. IEEE 4th International Workshop on Policies for Distributed Systems and Networks*, pp. 120–131. DOI: 10.1109/POLICY.2003.1206966.

[89] A. Kasten. "Secure semantic web data management". doctoralthesis. Universität Koblenz, Universitätsbibliothek, 2016, p. 280.

[90] K. Katkalov. "Ein modellgetriebener Ansatz zur Entwicklung informationsflusssicherer Systeme". doctoral thesis. Universität Augsburg, 2017.

[91] K. Katkalov, K. Stenzel, M. Borek, and W. Reif. "Model-Driven Development of Information Flow-Secure Systems with IFlow". In: *2013 International Conference on Social Computing*. IEEE, 2013, pp. 51–56. DOI: 10.1109/SocialCom.2013.14.

[92] S. Katsikeas, P. Johnsson, S. Hacks, and R. Lagerström. "VehicleLang: A probabilistic modeling and simulation language for modern vehicle IT infrastructures". In: *Computers & Security* 117 (June 1, 2022), p. 102705. ISSN: 0167-4048. DOI: 10.1016/j.cose.2022.102705.

[93] Y. R. Kirschner. "Model-Driven Reverse Engineering of Technology-Induced Architecture for Quality Prediction". In: *European Conference on Software Architecture (ECSA) Workshop Proceedings.* Vol. 2978. CEUR-WS.org, 2021. URL: http://ceur-ws.org/Vol-2978/ds-paper100.pdf.

[94] Y. R. Kirschner, M. Walter, F. Bossert, R. Heinrich, and A. Koziolek. “Automatic Derivation of Vulnerability Models for Software Architectures”. In: *2023 IEEE 20th International Conference on Software Architecture Companion (ICSA-C)*. MDE4SA – 3rd International Workshop On Model-driven Engineering for Software Architecture. IEEE, 2023, pp. 276–283. DOI: 10.1109/ICSA-C57050.2023.00065.

[95] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. “Spectre Attacks: Exploiting Speculative Execution”. In: *40th IEEE Symposium on Security and Privacy (S&P’19)*. IEEE, 2019. DOI: 10.1109/SP.2019.00002.

[96] L. Kohnfelder and P. Garg. “The threats to our products”. In: (1999). URL: https://www.first.org/global/sigs/cti/curriculum/The-Threats-To-Our-Products.docx (visited on 03/29/2023).

[97] M. Konersmann, A. Kaplan, T. Kühn, R. Heinrich, A. Koziolek, R. Reussner, J. Jürjens, M. al-Doori, N. Boltz, M. Ehl, D. Fuchs, K. Groser, S. Hahner, J. Keim, M. Lohr, T. Sağlam, S. Schulz, and J.-P. Töberg. “Evaluation Methods and Replicability of Software Architecture Research Objects”. In: *2022 IEEE 19th International Conference on Software Architecture (ICSA)*. Mar. 2022, pp. 157–168. DOI: 10.1109/ICSA53651.2022.00023.

[98] B. Kordy, L. Piètre-Cambacédès, and P. Schweitzer. “DAG-based attack and defense modeling: Don’t miss the forest for the attack trees”. In: *Computer Science Review* 13–14 (Nov. 2014), pp. 1–38. ISSN: 15740137. DOI: 10.1016/j.cosrev.2014.07.001.

[99] M. E. Kramer, M. Hecker, S. Greiner, K. Bao, and K. Yurchenko. *Model-Driven Specification and Analysis of Confidentiality in Component-Based Systems*. Tech. rep. 2017,12. Karlsruhe: Department of Informatics, Karlsruhe Institute of Technology, Dec. 2017. DOI: 10.5445/IR/1000076957.

[100] M. Lehman. “Programs, life cycles, and laws of software evolution”. In: *Proceedings of the IEEE* 68.9 (1980), pp. 1060–1076. DOI: 10.1109/PROC.1980.11805.

[101] L. Lemaire, J. Vossaert, B. De Decker, and V. Naessens. “Security evaluation of cyber-physical systems using automatically generated attack trees”. In: *Critical Information Infrastructures Security: 12th*

International Conference, CRITIS. Vol. 10707. Lecture Notes in Computer Science. Springer, 2018, pp. 225–228. DOI: 10.1007/978-3-319-99843-5_20.

[102] M. Levandowsky and D. Winter. “Distance between sets”. In: *Nature* 234.5323 (1971), pp. 34–35. DOI: 10.1038/234034a0.

[103] X. Li, X. Chang, J. A. Board, and K. S. Trivedi. “A novel approach for software vulnerability classification”. In: *2017 Annual Reliability and Maintainability Symposium (RAMS)*. 2017 Annual Reliability and Maintainability Symposium (RAMS). Jan. 2017, pp. 1–7. DOI: 10.1109/RAM.2017.7889792.

[104] D. Lin, P. Rao, E. Bertino, N. Li, and J. Lobo. “EXAM: a comprehensive environment for the analysis of access control policies”. In: *International Journal of Information Security* 9.4 (Aug. 1, 2010), pp. 253–273. ISSN: 1615-5270. DOI: 10.1007/s10207-010-0106-1.

[105] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, M. Hamburg, and R. Strackx. “Meltdown: Reading Kernel Memory from User Space”. In: vol. 63. 6. ACM, May 2020, pp. 46–56. DOI: 10.1145/3357033.

[106] T. Lodderstedt, D. Basin, and J. Doser. “SecureUML: A UML-Based Modeling Language for Model-Driven Security”. In: *UML 2002 — The Unified Modeling Language*. Vol. 24. Springer, Berlin, Heidelberg, 2002, pp. 426–441. ISBN: 978-3-540-45800-5. DOI: 10.1007/3-540-45800-X_33.

[107] R. Martin. *Agile Software Development: Principles, Patterns, and Practices*. Alan Apt series. Pearson Education, 2003. ISBN: 9780135974445.

[108] E. Martins and M. Pascoal. “A new implementation of Yen’s ranking loopless paths algorithm”. en. In: *Quarterly Journal of the Belgian, French and Italian Operations Research Societies* 1.2 (June 2003). ISSN: 1619-4500. DOI: 10.1007/s10288-002-0010-2.

[109] S. Mauw and M. Oostdijk. “Foundations of Attack Trees”. In: *Information Security and Cryptology - ICISC 2005*. Ed. by D. H. Won and S. Kim. Vol. 3935. Lecture Notes in Computer Science. Springer, 2006, pp. 186–198. ISBN: 978-3-540-33354-8. DOI: 10.1007/11734727_17.

[110] G. McGraw. *Software Security - Building Security In*. Addison-Wesley Professional, 2006. ISBN: 0-321-35670-5.

[111] MITRE Corporation. *CWE*. URL: https://cwe.mitre.org/ (visited on 10/25/2021).

[112] MITRE Corporation. *CWE - Common Weakness Scoring System (CWSS)*. URL: https://cwe.mitre.org/cwss/cwss_v1.0.1.html (visited on 10/25/2021).

[113] MITRE Corporation. *CWE-20: Improper Input Validation*. URL: https://cwe.mitre.org/data/definitions/20.html (visited on 06/14/2023).

[114] MITRE Corporation. *CWE-312*. URL: https://cwe.mitre.org/data/definitions/312.html (visited on 10/25/2021).

[115] MITRE Corporation. *CWE-732*. URL: https://cwe.mitre.org/data/definitions/732.html (visited on 02/06/2023).

[116] MITRE Corporation. *CWE-798: Use of Hard-coded Credentials*. URL: https://cwe.mitre.org/data/definitions/798.html (visited on 11/21/2022).

[117] MITRE Corporation. *CWE-922*. URL: https://cwe.mitre.org/data/definitions/922.html (visited on 10/25/2021).

[118] A. K. Y. S. Mohamed, D. Auer, D. Hofer, and J. Küng. "A systematic literature review for authorization and access control: definitions, strategies and models". In: *International Journal of Web Information Systems* 18.2 (Jan. 1, 2022), pp. 156–180. ISSN: 1744-0084. DOI: 10.1108/IJWIS-04-2022-0077.

[119] D. Monschein, M. Mazkatli, R. Heinrich, and A. Koziolek. "Enabling Consistency between Software Artefacts for Software Adaption and Evolution". In: *International Conference on Software Architecture (ICSA)*. IEEE, 2021, pp. 1–12. DOI: 10.1109/ICSA51549.2021.00009.

[120] M. Munier, V. Lalanne, and M. Ricarde. "Self-Protecting Documents for Cloud Storage Security". In: *11th International Conference on Trust, Security and Privacy in Computing and Communications*. IEEE, June 2012, pp. 1231–1238. DOI: 10.1109/TrustCom.2012.261.

[121] M. Nabeel. "The Many Faces of End-to-End Encryption and Their Security Analysis". In: *2017 IEEE International Conference on Edge Computing (EDGE)*. IEEE, 2017, pp. 252–259. DOI: 10.1109/IEEE.EDGE.2017.47.

[122] S. Newman. *Building Microservices : Designing Fine-Grained Systems*. O'Reilly Media, Incorporated, 2015. ISBN: 978-1-4919-5033-3.

[123] P. Nguyen, M. Kramer, J. Klein, and Y. L. Traon. "An extensive systematic review on the Model-Driven Development of secure systems". In: *Information and Software Technology* 68 (Dec. 2015), pp. 62–81. ISSN: 09505849. DOI: 10.1016/j.infsof.2015.08.006.

[124] Q. Ni, E. Bertino, J. Lobo, and S. B. Calo. "Privacy-Aware Role-Based Access Control". In: *IEEE Security & Privacy Magazine* 7.4 (July 2009), pp. 35–43. ISSN: 1540-7993. DOI: 10.1109/MSP.2009.102. (Visited on 08/22/2019).

[125] NIST. *cve-2009-0783*. URL: https://nvd.nist.gov/vuln/detail/cve-2009-0783 (visited on 06/15/2023).

[126] NIST. *CVE-2014-1761*. URL: https://nvd.nist.gov/vuln/detail/CVE-2014-1761 (visited on 10/25/2021).

[127] NIST. *CVE-2017-5715 Detail*. URL: https://nvd.nist.gov/vuln/detail/CVE-2017-5715 (visited on 03/14/2022).

[128] NIST. *CVE-2021-22879 Detail*. URL: https://nvd.nist.gov/vuln/detail/CVE-2021-22879 (visited on 03/21/2022).

[129] NIST. *CVE-2021-28374*. URL: https://nvd.nist.gov/vuln/detail/CVE-2021-28374 (visited on 10/25/2021).

[130] NIST. *CVE-2021-44228 Detail*. URL: https://nvd.nist.gov/vuln/detail/CVE-2021-44228 (visited on 03/30/2022).

[131] *NVD*. URL: https://nvd.nist.gov/vuln (visited on 10/25/2021).

[132] OASIS Open. *eXtensible Access Control Markup Language (XACML) Version 3.0*. Jan. 22, 2013. URL: https://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.html (visited on 04/06/2022).

[133] OASIS Open. *XACML v3.0 Administration and Delegation Profile Version 1.0*. Nov. 13, 2014. URL: https://docs.oasis-open.org/xacml/3.0/xacml-3.0-administration-v1-spec-en.html (visited on 03/22/2023).

[134] OASIS Open. *XML schema XACML*. Jan. 22, 2013. URL: http://docs.oasis-open.org/xacml/3.0/xacml-core-v3-schema-wd-17.xsd (visited on 04/06/2022).

[135] A. L. Opdahl and G. Sindre. "Experimental comparison of attack trees and misuse cases for security threat identification". In: *Information and Software Technology* 51.5 (May 2009), pp. 916–932. ISSN: 09505849. DOI: 10.1016/j.infsof.2008.05.013.

[136] OWASP. *Software Assurance Maturity Model.* URL: https://owaspsamm.org/guidance/quick-start-guide/ (visited on 10/13/2022).

[137] *OWASP Top Ten Web Application Security Risks | OWASP.* URL: https://owasp.org/www-project-top-ten/ (visited on 10/25/2021).

[138] J. Park and R. Sandhu. " The UCONABC usage control model ". In: *ACM Transactions on Information and System Security* 7.1 (July 2004), pp. 128–174. ISSN: 10949224. DOI: 10.1145/984334.984339.

[139] A. Paverd, A. Martin, and I. Brown. "Modelling and automatically analysing privacy properties for honest-but-curious adversaries". In: (2014). URL: https://www.cs.ox.ac.uk/people/andrew.paverd/casper/casper-privacy-report.pdf (visited on 07/31/2023).

[140] T. R. Peltier. "Social Engineering: Concepts and Solutions". In: *Information Systems Security* 15.5 (2006), pp. 13–21. DOI: 10.1201/1086.1065898X/46353.15.4.20060901/95427.3.

[141] R. Peters. *cron.* Springer, 2009, pp. 81–85. ISBN: 9781430218418. DOI: 10.1007/978-1-4302-1842-5_12.

[142] C. Phillips and L. P. Swiler. "A graph-based system for network-vulnerability analysis". In: *Proceedings of the 1998 workshop on New security paradigms.* NSPW98: New Security Paradigms Workshop. Charlottesville Virginia USA: ACM, Jan. 1998, pp. 71–79. ISBN: 978-1-58113-168-0. DOI: 10.1145/310889.310919.

[143] R. Pilipchuk. "Architectural Alignment of Access Control Requirements Extracted from Business Processes". PhD thesis. Karlsruher Institut für Technologie (KIT), 2021. 258 pp. DOI: 10.5445/IR/1000140856.

[144] A. M. Pirca and H. S. Lallie. "An empirical evaluation of the effectiveness of attack graphs and MITRE ATT&CK matrices in aiding cyber attack perception amongst decision-makers". In: *Computers & Security* 130 (2023), p. 103254. DOI: 10.1016/j.cose.2023.103254.

[145] M. Plachkinova and C. Maurer. "*Teaching Case* Security Breach at Target". In: *Journal of Information Systems Education* 29.1 (2018), pp. 11–20. URL: https://jise.org/Volume29/n1/JISEv29n1p11.html (visited on 06/29/2023).

[146] V. S. Pless. "Encryption schemes for computer confidentiality". In: *IEEE Transactions on Computers* 26.11 (1977), pp. 1133–1136. DOI: 10.1109/TC.1977.1674759.

[147] N. Polatidis, M. Pavlidis, and H. Mouratidis. “Cyber-attack path discovery in a dynamic supply chain maritime risk management system”. In: *Computer Standards & Interfaces* 56 (2018), pp. 74–82. ISSN: 0920-5489. DOI: 10.1016/j.csi.2017.09.006.

[148] N. Polatidis, E. Pimenidis, M. Pavlidis, S. Papastergiou, and H. Mouratidis. “From product recommendation to cyber-attack prediction: generating attack graphs and predicting future attacks”. In: *Evolving Systems* 11.3 (Sept. 2020), pp. 479–490. ISSN: 1868-6478, 1868-6486. DOI: 10.1007/s12530-018-9234-z.

[149] C. Ponikwar, H. Hof, S. Gopinath, and L. Wischhof. “Beyond the Dolev-Yao Model: Realistic Application-Specific Attacker Models for Applications Using Vehicular Communication”. In: *CoRR* abs/1607.08277 (2016). arXiv: 1607.08277. URL: http://arxiv.org/abs/1607.08277.

[150] A. Pretschner, M. Hilty, and D. Basin. “Distributed usage control”. In: *Communications of the ACM* 49.9 (2006), pp. 39–44. ISSN: 0001-0782. DOI: 10.1145/1151030.1151053.

[151] *Prioritization to Prediction Volume 8: Measuring and Minimizing Exploitability.* Cyentia Institute, Kenna Security, 2022. URL: https://library.cyentia.com/report/report_008756.html (visited on 05/02/2023).

[152] G. Rasner. *Cybersecurity and third-party risk third party threat hunting.* Wiley Data and Cybersecurity, 2021. ISBN: 9781119809562.

[153] *Report: IBM Security X-Force Threat Intelligence Index 2023.* en-us. Mar. 2023. URL: https://web.archive.org/web/20230320024631/https://www.ibm.com/downloads/cas/DB4GL8YM (visited on 04/04/2023).

[154] R. Reussner, S. Becker, J. Happe, R. Heinrich, A. Koziolek, H. Koziolek, M. Kramer, and K. Krogmann. *Modeling and Simulating Software Architectures – The Palladio Approach.* Cambridge, MA: MIT Press, Oct. 2016. 408 pp. ISBN: 9780262034760. URL: https://web.archive.org/web/20180415104041/http://mitpress.mit.edu/books/modeling-and-simulating-software-architectures.

[155] K. Rostami, R. Heinrich, A. Busch, and R. Reussner. “Architecture-Based Change Impact Analysis in Information Systems and Business Processes”. In: *International Conference on Software Architecture (ICSA).* 2017, pp. 179–188. DOI: 10.1109/ICSA.2017.17.

[156] K. Rostami, J. Stammel, R. Heinrich, and R. Reussner. "Architecture-Based Assessment and Planning of Change Requests". In: *Proceedings of the 11th International ACM SIGSOFT Conference on Quality of Software Architectures*. QoSA '15. ACM, 2015, pp. 21–30. ISBN: 9781450334709. DOI: 10.1145/2737182.2737198.

[157] P. Runeson and M. Höst. "Guidelines for conducting and reporting case study research in software engineering". In: *Empirical Software Engineering* 14.2 (Dec. 19, 2008), p. 131. ISSN: 1573-7616. DOI: 10.1007/s10664-008-9102-8.

[158] M. Sabt, M. Achemlal, and A. Bouabdallah. "Trusted Execution Environment: What It is, and What It is Not". In: *2015 IEEE Trustcom/BigDataSE/ISPA*. IEEE, pp. 57–64. DOI: 10.1109/Trustcom.2015.357.

[159] J. Saltzer and M. Schroeder. "The protection of information in computer systems". In: *Proceedings of the IEEE* 63.9 (1975), pp. 1278–1308. DOI: 10.1109/PROC.1975.9939.

[160] S. van Schaik, A. Kwong, D. Genkin, and Y. Yarom. *SGAxe: How SGX Fails in Practice*. 2020. URL: https://sgaxeattack.com/ (visited on 09/01/2021).

[161] R. Schmidt, M. Möhring, R.-C. Härting, C. Reichstein, P. Neumaier, and P. Jozinović. "Industry 4.0 - Potentials for Creating Smart Products: Empirical Research Results". In: *Business Information Systems*. Ed. by W. Abramowicz. Vol. 208. Series Title: Lecture Notes in Business Information Processing. Cham: Springer, 2015, pp. 16–27. DOI: 10.1007/978-3-319-19027-3_2.

[162] B. Schneier. "Attack trees". In: *Dr. Dobb's journal* 24.12 (1999), pp. 21–29. URL: https://www.schneier.com/academic/archives/1999/12/attack_trees.html (visited on 03/29/2023).

[163] S. Schulz, F. Reiche, S. Hahner, and J. Schiffl. "Continuous Secure Software Development and Analysis". In: *Symposium on Software Performance 2021*. 12th Symposium on Software Performance. SSP 2021 (Nov. 9–10, 2021). Vol. 3043. CEUR Workshop Proceedings. 46.23.01; LK 01. RWTH Aachen, 2021. URL: https://ceur-ws.org/Vol-3043/short7.pdf (visited on 07/20/2023).

[164] *Securing the Software Supply Chain: Recommended Practices Guide for Developers.* Cybersecurity and Infrastructure Security Agency (CISA), Aug. 2022, p. 64. URL: https://www.cisa.gov/sites/default/files/publications/ESF_SECURING_THE_SOFTWARE_SUPPLY_CHAIN_DEVELOPERS.PDF (visited on 04/28/2023).

[165] S. Seifermann, R. Heinrich, D. Werle, and R. Reussner. "A unified model to detect information flow and access control violations in software architectures". In: *Proceedings of the 18th International Conference on Security and Cryptography, SECRYPT 2021.* 18th International Conference on Security and Cryptography. SciTePress, July 2021, pp. 26–37. ISBN: 978-9897585241. DOI: 10.5220/0010515300260037.

[166] S. Seifermann. "Architectural Data Flow Analysis for Detecting Violations of Confidentiality Requirements". PhD thesis. Karlsruher Institut für Technologie (KIT), 2022. 282 pp. DOI: 10.5445/IR/1000148748.

[167] S. Seifermann, R. Heinrich, and R. Reussner. "Data-Driven Software Architecture for Analyzing Confidentiality". In: *International Conference on Software Architecture (ICSA).* Hamburg, Germany: IEEE, Mar. 2019, pp. 1–10. ISBN: 978-1-72810-528-4. DOI: 10.1109/ICSA.2019.00009.

[168] S. Seifermann, R. Heinrich, D. Werle, and R. Reussner. "Detecting Violations of Access Control and Information Flow Policies in Data Flow Diagrams". In: *Journal of Systems and Software* 184 (2021), p. 111138. ISSN: 0164-1212. DOI: 10.1016/j.jss.2021.111138.

[169] S. Seifermann and M. Walter. "Evolving a use case for industry 4.0 environments towards integration of physical access control". In: *Workshops of the Software Engineering Conference.* Fachtagungen "Software Engineering" - "Software Management". SE 2019 - SWM 2019 (Feb. 18–22, 2019). Ed. by M. Konersmann. Vol. 2308. CEUR Workshop Proceedings, 2019, pp. 106–108. DOI: 10.5445/IR/1000092827. URL: https://ceur-ws.org/Vol-2308/emls2019paper03.pdf.

[170] E. I. Sharing and A. C. (E-ISAC). *Analysis of the cyber attack on the Ukrainian power grid, Defense Use Case.* Tech. rep. 2016, pp. 1–29.

[171] O. Sheyner, J. Haines, S. Jha, R. Lippmann, and J. Wing. "Automated generation and analysis of attack graphs". In: *Proceedings 2002 IEEE Symposium on Security and Privacy.* IEEE, May 2002, pp. 273–284. DOI: 10.1109/SECPRI.2002.1004377.

[172] A. Shostack. *Threat modeling: Designing for security*. John Wiley & Sons, 2014. ISBN: 9781118809990.

[173] X. Shu, K. Tian, A. Ciambrone, and D. Yao. "Breaking the Target: An Analysis of Target Data Breach and Lessons Learned". In: (Jan. 17, 2017). arXiv: 1701.04940. URL: http://arxiv.org/abs/1701.04940 (visited on 08/30/2021).

[174] F. Shull, V. Basili, B. Boehm, A. Brown, P. Costa, M. Lindvall, D. Port, I. Rus, R. Tesoriero, and M. Zelkowitz. "What we have learned about fighting defects". In: *Proceedings Eighth IEEE Symposium on Software Metrics*. June 2002, pp. 249–258. DOI: 10.1109/METRIC.2002.1011343.

[175] C. E. da Silva, J. D. S. da Silva, C. Paterson, and R. Calinescu. "Self-adaptive Role-based Access Control for Business Processes". In: *Proceedings of the 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 193–203. ISBN: 978-1-5386-1550-8. DOI: 10.1109/SEAMS.2017.13. URL: https://doi.org/10.1109/SEAMS.2017.13 (visited on 08/28/2019).

[176] G. Sindre. "Mal-Activity Diagrams for Capturing Attacks on Business Processes". In: *Requirements Engineering: Foundation for Software Quality, 13th International Working Conference, REFSQ*. Vol. 4542. Lecture Notes in Computer Science. Springer, 2007, pp. 355–366. ISBN: 978-3-540-73030-9. DOI: 10.1007/978-3-540-73031-6_27.

[177] G. Sindre and A. L. Opdahl. "Eliciting security requirements with misuse cases". In: *Requirements Engineering* 10.1 (Jan. 2005), pp. 34–44. ISSN: 0947-3602, 1432-010X. DOI: 10.1007/s00766-004-0194-4.

[178] L. Sion, D. Van Landuyt, K. Yskout, and W. Joosen. "SPARTA: Security & Privacy Architecture Through Risk-Driven Threat Assessment". In: *2018 IEEE International Conference on Software Architecture Companion, (ICSA-C)*. IEEE, 2018, pp. 89–92. DOI: 10.1109/ICSA-C.2018.00032.

[179] L. Sion, K. Yskout, D. Van Landuyt, and W. Joosen. "Solution-Aware Data Flow Diagrams for Security Threat Modeling". In: *Proceedings of the 33rd Annual ACM Symposium on Applied Computing, (SAC)*. ACM, 2018, pp. 1425–1432. ISBN: 9781450351911. DOI: 10.1145/3167132.3167285.

[180] N. Slimani, H. Khambhammettu, K. Adi, and L. Logrippo. "UACML: Unified Access Control Modeling Language". In: *2011 4th IFIP International Conference on New Technologies, Mobility and Security*. 2011 4th IFIP International Conference on New Technologies, Mobility and Security. Feb. 2011, pp. 1–8. DOI: 10.1109/NTMS.2011.5721143.

[181] T. Sommestad. "A framework and theory for cyber security assessments". Stockholm. KTH Royal Institute of Technology, 2012. 248 pp. ISBN: 9789175015118.

[182] T. Sommestad, M. Ekstedt, and H. Holm. "The Cyber Security Modeling Language: A Tool for Assessing the Vulnerability of Enterprise System Architectures". In: *IEEE SYSTEMS JOURNAL* 7.3 (2013), p. 11. DOI: 10.1109/JSYST.2012.2221853.

[183] *Source Code Analysis Tools | OWASP Foundation*. URL: https://owasp.org/www-community/Source_Code_Analysis_Tools (visited on 06/05/2023).

[184] J. Spring, E. Hatleback, A. Householder, A. Manion, and D. Shick. "Time to Change the CVSS?" In: *IEEE Security & Privacy* 19.2 (Mar. 2021), pp. 74–78. ISSN: 1540-7993, 1558-4046. DOI: 10.1109/MSEC.2020.3044475.

[185] J. Spring, E. Hatleback, A. Manion, and D. Shic. *Towards improving CVSS*. Tech. rep. Software Engineering Institute, Carnegie Mellon University, 2018.

[186] J. M. Spring, E. Hatleback, A. D. Householder, A. Manion, M. Oliver, V. Sarvapalli, D. Shick, and L. Tyzenhaus. *Prioritizing vulnerability response: A stakeholder-specific vulnerability categorization (version 2.0)*. Software Engineering Institute, Carnegie Mellon University, Apr. 2021, p. 68. URL: https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=653459 (visited on 03/28/2023).

[187] H. Stachowiak. *Allgemeine Modelltheorie*. Springer, 1973. ISBN: 3-211-81106-0.

[188] T. Stahl, M. Völter, and K. Czarnecki. *Model-driven software development: technology, engineering, management*. John Wiley & Sons, Inc., 2006, p. 448. ISBN: 978-1-118-72576-4.

[189] J. Stammel and R. Reussner. "Kamp: Karlsruhe architectural maintainability prediction". In: *Proceedings of the 1. Workshop des GI-Arbeitskreises Langlebige Softwaresysteme (L2S2):" Design for Future-Langlebige Softwaresysteme*. 2009, pp. 87–98.

[190] J. J. Stammel. "Architekturbasierte Bewertung und Planung von Änderungsanfragen". PhD thesis. Karlsruher Institut für Technologie (KIT), 2015. DOI: 10.5445/IR/1000053953. (Visited on 04/13/2023).

[191] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro. *EMF: Eclipse Modeling Framework*. Eclipse Series. Pearson Education, 2008. ISBN: 9780132702218.

[192] M. Strittmatter. "A Reference Structure for Modular Metamodels of Quality-Describing Domain-Specific Modeling Languages". PhD thesis. Karlsruher Institut für Technologie (KIT), 2020. 482 pp. ISBN: 978-3-7315-0982-0. DOI: 10.5445/KSP/1000098906.

[193] S. R. Subramanya and B. K. Yi. "Digital rights management". In: *IEEE Potentials* 25.2 (Mar. 2006), pp. 31–34. ISSN: 0278-6648. DOI: 10.1109/MP.2006.1649008.

[194] L. SWEENEY. "k-ANONYMITY: A MODEL FOR PROTECTING PRIVACY". In: *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems* 10.05 (2002), pp. 557–570. DOI: 10.1142/S0218488502001648.

[195] E. Taspolatoglu and R. Heinrich. "Context-Based Architectural Security Analysis". In: *Working IEEE/IFIP Conference on Software Architecture (WICSA)*. IEEE, Apr. 2016, pp. 281–282. ISBN: 978-1-5090-2131-4. DOI: 10.1109/WICSA.2016.55.

[196] I. A. Tøndel, J. Jensen, and L. Røstad. "Combining Misuse Cases with Attack Trees and Security Activity Models". In: *2010 International Conference on Availability, Reliability and Security (ARES)*. IEEE, Feb. 2010, pp. 438–445. ISBN: 978-1-4244-5879-0. DOI: 10.1109/ARES.2010.101.

[197] K. Toumi, C. Andrés, and A. Cavalli. "Trust-orBAC: A Trust Access Control Model in Multi-Organization Environments". In: *Information Systems Security, 8th International Conference, (ICISS)*. Ed. by V. Venkatakrishnan and D. Goswami. Vol. 7671. Lecture Notes in Computer Science. Springer, 2012, pp. 89–103. ISBN: 978-3-642-35129-7. DOI: 10.1007/978-3-642-35130-3_7. (Visited on 03/30/2021).

[198] A. Tria and H. Choukri. "Invasive Attacks". In: *Encyclopedia of Cryptography and Security*. Boston, MA: Springer US, 2011, pp. 623–629. ISBN: 978-1-4419-5906-5. DOI: 10.1007/978-1-4419-5906-5_511.

[199] K. Tuma, S. Peldszus, D. Strüber, R. Scandariato, and J. Jürjens. "Checking security compliance between models and code". In: *Software and Systems Modeling* 22.1 (Feb. 1, 2023), pp. 273–296. ISSN: 1619-1374. DOI: 10.1007/s10270-022-00991-5.

[200] K. Tuma, R. Scandariato, and M. Balliu. "Flaws in Flows: Unveiling Design Flaws via Information Flow Analysis". In: *International Conference on Software Architecture (ICSA)*. IEEE, 2019, pp. 191–200. DOI: 10.1109/ICSA.2019.00028.

[201] K. Tuma, L. Sion, R. Scandariato, and K. Yskout. "Automating the Early Detection of Security Design Flaws". In: *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*. MODELS '20. ACM, 2020, pp. 332–342. ISBN: 978-1-4503-7019-6. DOI: 10.1145/3365438.3410954. URL: https://doi.org/10.1145/3365438.3410954.

[202] F. Turkmen, J. den Hartog, S. Ranise, and N. Zannone. "Analysis of XACML Policies with SMT". In: *Principles of Security and Trust*. Springer, 2015, pp. 115–134. ISBN: 978-3-662-46666-7. DOI: 10.1007/978-3-662-46666-7_7.

[203] C. Van Rijsbergen and C. Van Rijsbergen. *Information Retrieval*. Butterworths, 1979. ISBN: 9780408709293.

[204] D. Verma, E. Bertino, G. de Mel, and J. Melrose. "On the Impact of Generative Policies on Security Metrics". In: *2019 IEEE International Conference on Smart Computing (SMARTCOMP)*. IEEE, June 2019, pp. 104–109. ISBN: 978-1-72811-689-1. DOI: 10.1109/SMARTCOMP.2019.00037.

[205] S. Vimercati. "Discretionary Access Control Policies (DAC)". In: *Encyclopedia of Cryptography and Security*. Boston, MA: Springer US, 2011, pp. 356–358. ISBN: 978-1-4419-5906-5. DOI: 10.1007/978-1-4419-5906-5_817.

[206] S. Vimercati and P. Samarati. "Mandatory Access Control Policy (MAC)". In: *Encyclopedia of Cryptography and Security*. Boston, MA: Springer US, 2011, pp. 758–758. ISBN: 978-1-4419-5906-5. DOI: 10.1007/978-1-4419-5906-5_822.

[207] W3C. *XML Path Language (XPath) 3.1.* Mar. 21, 2017. URL: https://www.w3.org/TR/xpath-31/ (visited on 01/13/2023).

[208] M. Walter. *Dataset: Context-based Access Control and Attack Modelling and Analysis.* 2023. DOI: 10.5281/zenodo.8208073.

[209] M. Walter, S. Hahner, T. Bureš, P. Hnětynka, R. Heinrich, and R. Reussner. "Architecture-based attack propagation and variation analysis for identifying confidentiality issues in Industry 4.0". In: *at - Automatisierungstechnik* 71.6 (2023), pp. 443–452. DOI: 10.1515/auto-2022-0135.

[210] M. Walter, S. Hahner, S. Seifermann, T. Bures, P. Hnetynka, J. Pacovský, and R. Heinrich. "Architectural Optimization for Confidentiality Under Structural Uncertainty". In: *Software Architecture : 15th European Conference, ECSA 2021 Tracks and Workshops.* 15th European Conference on Software Architecture. ECSA 2021 (Online, Sept. 13–17, 2021). Vol. 13365. Lecture Notes in Computer Science. Springer, 2022, pp. 309–332. ISBN: 978-3-031-15115-6. DOI: 10.1007/978-3-031-15116-3_14.

[211] M. Walter, R. Heinrich, and R. Reussner. "Architectural Attack Propagation Analysis for Identifying Confidentiality Issues". In: *2022 IEEE 19th International Conference on Software Architecture (ICSA).* IEEE, Mar. 2022, pp. 1–12. ISBN: 978-1-66541-728-0. DOI: 10.1109/ICSA53651.2022.00009.

[212] M. Walter, R. Heinrich, and R. Reussner. "Architecture-based Attack Path Analysis for Identifying Potential Security Incidents". In: *Software Architecture.* 17th European Conference on Software Architecture (ECSA). ECSA 2023. 2023. DOI: 10.1007/978-3-031-42592-9_3.

[213] M. Walter, R. Heinrich, and R. Reussner. "Identifizierung von Vertraulichkeitsproblemen mithilfe von Angriffsausbreitung auf Architektur". German. In: *Software Engineering 2023. Hrsg.: R. Engels.* Software Engineering. SE 2023 (Paderborn, Deutschland, Feb. 20–24, 2023). Vol. 332. GI-Edition : Lecture Notes in Informatics / Proceedings. 46.23.03; LK 01. Gesellschaft für Informatik (GI), 2023, pp. 123–124. ISBN: 978-3-88579-726-5.

[214] M. Walter and R. Reussner. "Tool-based Attack Graph Estimation and Scenario Analysis for Software Architectures". In: *Software Architecture. ECSA 2022 Tracks and Workshops.* 16th European Conference on Software Architecture. ECSA 2022. Ed. by T. Batista, T. Bureš, C.

Raibulet, and H. Muccini. Springer International Publishing, 2023, pp. 45–61. DOI: 10.1007/978-3-031-36889-9_5.

[215] M. Walter, S. Seifermann, and R. Heinrich. "A Taxonomy of Dynamic Changes Affecting Confidentiality". In: *11th Workshop Design For Future – Langlebige Softwaresysteme*. 11. Workshop "Design For Future - Langlebige Softwaresysteme" (DFF 2020) - 22. Workshop Software-Reeingineering and Evolution. 2020.

[216] H. Wang, Z. Chen, J. Zhao, X. Di, and D. Liu. "A Vulnerability Assessment Method in Industrial Internet of Things Based on Attack Graph and Maximum Flow". In: *IEEE Access* 6 (2018), pp. 8599–8609. DOI: 10.1109/ACCESS.2018.2805690.

[217] D. Werle, S. Seifermann, and A. Koziolek. "Data Stream Operations as First-Class Entities in Component-Based Performance Models". In: *Software Architecture European Conference on Software Architecture (ECSA)*. Ed. by A. Jansen, I. Malavolta, H. Muccini, I. Ozkaya, and O. Zimmermann. Cham: Springer International Publishing, 2020, pp. 148–164. ISBN: 978-3-030-58923-3. DOI: 10.1007/978-3-030-58923-3_10.

[218] W. Wideł, M. Audinot, B. Fila, and S. Pinchinat. "Beyond 2014: Formal Methods for Attack Tree–based Security Modeling". In: *ACM Computing Surveys* 52.4 (Aug. 30, 2019), 75:1–75:36. ISSN: 0360-0300. DOI: 10.1145/3331524.

[219] W. Wideł, S. Hacks, M. Ekstedt, P. Johnson, and R. Lagerström. "The meta attack language - a formal description". In: *Computers & Security* 130 (July 1, 2023), p. 103284. ISSN: 0167-4048. DOI: 10.1016/j.cose.2023.103284.

[220] P. A. Wortman and J. Chandy. "Translation of AADL model to security attack tree (TAMSAT) to SMART evaluation of monetary security risk". In: *Information Security Journal: A Global Perspective* 32.4 (Aug. 4, 2022), pp. 1–17. ISSN: 1939-3555. DOI: 10.1080/19393555.2022.2106909.

[221] Z. B. Yahya, F. B. Ktata, and K. Ghedira. "Multi-organizational Access Control Model Based on Mobile Agents for Cloud Computing". In: *2016 IEEE/WIC/ACM International Conference on Web Intelligence (WI)*. Omaha, NE, USA: IEEE, Oct. 2016, pp. 656–659. ISBN: 978-1-5090-4470-2. DOI: 10.1109/WI.2016.0116.

[222] T. Yoshizawa, D. Singelée, J. T. Muehlberg, S. Delbruel, A. Taherkordi, D. Hughes, and B. Preneel. "A Survey of Security and Privacy Issues in V2X Communication Systems". In: *ACM Comput. Surv.* 55.9 (Jan. 2023). ISSN: 0360-0300. DOI: 10.1145/3558052.

[223] B. Yuan, Z. Pan, F. Shi, and Z. Li. "An Attack Path Generation Methods Based on Graph Database". In: *2020 IEEE 4th Information Technology, Networking, Electronic and Automation Control Conference (ITNEC).* Vol. 1. IEEE, 2020, pp. 1905–1910. DOI: 10.1109/ITNEC48623.2020.9085039.

[224] K. Zenitani. "Attack graph analysis: An explanatory guide". In: *Computers & Security* 126 (Mar. 1, 2023), p. 103081. ISSN: 0167-4048. DOI: 10.1016/j.cose.2022.103081.

[225] G. Zhang and M. Parashar. "Context-aware dynamic access control for pervasive applications". In: *Proceedings of the Communication Networks and Distributed Systems Modeling and Simulation Conference.* 2004, pp. 21–30.

[226] N. Zhang, M. Ryan, and D. P. Guelev. "Evaluating Access Control Policies Through Model Checking". In: *Information Security.* Ed. by J. Zhou, J. Lopez, R. H. Deng, and F. Bao. Lecture Notes in Computer Science. Springer, 2005, pp. 446–460. ISBN: 978-3-540-31930-6. DOI: 10.1007/11556992_32.

A. Evaluation Results Effort Reduction Targeted Attack Graph Analysis

Table A.1.: Evaluation results for **Q6**

Scenario	Aff.	Ratio	Size	Min	Max
Power-Grid-1 (Storage-Application)	4	0.21	19	0.11	0.26
Power-Grid-2 (WithVPNRights)	3	0.16	19		
Power-Grid-3 (CorporateNetwork)	4	0.21	19		
Power-Grid-4 (VPNBridgeExternal)	5	0.26	19		
Power-Grid-5 (Workstation 02)	4	0.21	19		
Power-Grid-6 (Workstation 01)	5	0.26	19		
Power-Grid-7 (DataCenter)	5	0.26	19		
Power-Grid-8 (CallCenter)	5	0.26	19		
Power-Grid-9 (ExternalVPNBridge)	4	0.21	19		
Power-Grid-10 (CallCenterApplication)	4	0.21	19		
Power-Grid-11 (WithoutVPNRights)	4	0.21	19		
Power-Grid-12 (DC)	4	0.21	19		
Power-Grid-13 (DCS)	5	0.26	19		
Power-Grid-14 (DMSClient)	2	0.11	19		
Target-1 (FTP-Component)	2	0.13	15	0.13	0.27
Target-2 (BusinessComponent)	2	0.13	15		
Target-3 (Database)	2	0.13	15		
Target-4 (Intranet)	2	0.13	15		
Target-5 (POS-Component2)	2	0.13	15		
Target-6 (POS-Component3)	2	0.13	15		
Target-7 (Storage-Server)	3	0.20	15		
Target-8 (BusinessServer)	3	0.20	15		

Table A.1.: Continued evaluation results for **Q6**

Scenario	**Aff.**	**Ratio**	**Size**	**Min**	**Max**
Target-9 (POS3)	3	0.20	15		
Target-10 (ExternalSupplier)	3	0.20	15		
Target-11 (POS1)	2	0.13	15		
Target-12 (POS2)	3	0.20	15		
Target-13 (Internet)	3	0.20	15		
Target-14 (SupplierMachine)	4	0.27	15		

Table A.1.: Continued evaluation results for **Q6**

Scenario	**Aff.**	**Ratio**	**Size**	**Min**	**Max**
CloudStorage-1	3	0.10	31		
CloudStorage-2	3	0.10	31		
CloudStorage-3	3	0.10	31		
CloudStorage-4	2	0.06	31		
CloudStorage-5	3	0.10	31		
CloudStorage-6	3	0.10	31		
CloudStorage-7	3	0.10	31	0.06	0.10
CloudStorage-8	3	0.10	31		
CloudStorage-9	3	0.10	31		
CloudStorage-10	3	0.10	31		
CloudStorage-11	3	0.10	31		
CloudStorage-12	3	0.10	31		
CloudStorage-13	3	0.10	31		
CloudStorage-14	3	0.10	31		
Maintenance-1 (StorageServer)	2	0.25	8		
Maintenance-2 (TerminalComponent)	4	0.50	8		
Maintenance-3 (ProductionStorage)	4	0.50	8	0.25	0.50
Maintenance-4 (ProductionNetwork)	4	0.50	8		
Maintenance-5 (MachineController)	4	0.50	8		
Maintenance-6 (MachineComponent)	4	0.50	8		
Travelplanner	2	0.20	10	0.20	0.20

The Table A.1 lists the effort reduction. The first column gives the scenario with the architectural element as a starting point. The second column lists the number of affected architectural elements, the third column the ratio, the fourth the size and the two last columns give the minimum and maximum of the ratio for a scenario.

The Karlsruhe Series on Software Design and Quality

ISSN 1867-0067

Band 1 **Steffen Becker**
Coupled Model Transformations for QoS Enabled Component-Based Software Design.
ISBN 978-3-86644-271-9

Band 2 **Heiko Koziolek**
Parameter Dependencies for Reusable Performance Specifications of Software Components.
ISBN 978-3-86644-272-6

Band 3 **Jens Happe**
Predicting Software Performance in Symmetric Multi-core and Multiprocessor Environments.
ISBN 978-3-86644-381-5

Band 4 **Klaus Krogmann**
Reconstruction of Software Component Architectures and Behaviour Models using Static and Dynamic Analysis.
ISBN 978-3-86644-804-9

Band 5 **Michael Kuperberg**
Quantifying and Predicting the Influence of Execution Platform on Software Component Performance.
ISBN 978-3-86644-741-7

Band 6 **Thomas Goldschmidt**
View-Based Textual Modelling.
ISBN 978-3-86644-642-7

Band 7 **Anne Koziolek**
Automated Improvement of Software Architecture Models for Performance and Other Quality Attributes.
ISBN 978-3-86644-973-2

Die Bände sind unter www.ksp.kit.edu als PDF frei verfügbar oder als Druckausgabe bestellbar.

Band 8 **Lucia Happe**
Configurable Software Performance Completions through Higher-Order Model Transformations.
ISBN 978-3-86644-990-9

Band 9 **Franz Brosch**
Integrated Software Architecture-Based Reliability Prediction for IT Systems.
ISBN 978-3-86644-859-9

Band 10 **Christoph Rathfelder**
Modelling Event-Based Interactions in Component-Based Architectures for Quantitative System Evaluation.
ISBN 978-3-86644-969-5

Band 11 **Henning Groenda**
Certifying Software Component Performance Specifications.
ISBN 978-3-7315-0080-3

Band 12 **Dennis Westermann**
Deriving Goal-oriented Performance Models by Systematic Experimentation.
ISBN 978-3-7315-0165-7

Band 13 **Michael Hauck**
Automated Experiments for Deriving Performance-relevant Properties of Software Execution Environments.
ISBN 978-3-7315-0138-1

Band 14 **Zoya Durdik**
Architectural Design Decision Documentation through Reuse of Design Patterns.
ISBN 978-3-7315-0292-0

Band 15 **Erik Burger**
Flexible Views for View-based Model-driven Development.
ISBN 978-3-7315-0276-0

Band 16 **Benjamin Klatt**
Consolidation of Customized Product Copies into Software Product Lines.
ISBN 978-3-7315-0368-2

Band 17 **Andreas Rentschler**
Model Transformation Languages with Modular Information Hiding.
ISBN 978-3-7315-0346-0

Band 18 **Omar-Qais Noorshams**
Modeling and Prediction of I/O Performance in Virtualized Environments.
ISBN 978-3-7315-0359-0

Band 19 **Johannes Josef Stammel**
Architekturbasierte Bewertung und Planung von Änderungsanfragen.
ISBN 978-3-7315-0524-2

Band 20 **Alexander Wert**
Performance Problem Diagnostics by Systematic Experimentation.
ISBN 978-3-7315-0677-5

Band 21 **Christoph Heger**
An Approach for Guiding Developers to Performance and Scalability Solutions.
ISBN 978-3-7315-0698-0

Band 22 **Fouad ben Nasr Omri**
Weighted Statistical Testing based on Active Learning and Formal Verification Techniques for Software Reliability Assessment.
ISBN 978-3-7315-0472-6

Band 23 **Michael Langhammer**
Automated Coevolution of Source Code and Software Architecture Models.
ISBN 978-3-7315-0783-3

Band 24 **Max Emanuel Kramer**
Specification Languages for Preserving Consistency between Models of Different Languages.
ISBN 978-3-7315-0784-0

Band 25 **Sebastian Michael Lehrig**
Efficiently Conducting Quality-of-Service Analyses by Templating Architectural Knowledge.
ISBN 978-3-7315-0756-7

Band 26 **Georg Hinkel**
Implicit Incremental Model Analyses and Transformations.
ISBN 978-3-7315-0763-5

Band 27 **Christian Stier**
Adaptation-Aware Architecture Modeling and Analysis of Energy Efficiency for Software Systems.
ISBN 978-3-7315-0851-9

Band 28 **Lukas Märtin**
Entwurfsoptimierung von selbst-adaptiven Wartungs-mechanismen für software-intensive technische Systeme.
ISBN 978-3-7315-0852-6

Band 29 **Axel Busch**
Quality-driven Reuse of Model-based Software Architecture Elements.
ISBN 978-3-7315-0951-6

Band 30 **Kiana Busch**
An Architecture-based Approach for Change Impact Analysis of Software-intensive Systems.
ISBN 978-3-7315-0974-5

Band 31 **Misha Strittmatter**
A Reference Structure for Modular Metamodels of Quality-Describing Domain-Specific Modeling Languages.
ISBN 978-3-7315-0982-0

Die Bände sind unter www.ksp.kit.edu als PDF frei verfügbar oder als Druckausgabe bestellbar.

Band 32 **Markus Frank**
Model-Based Performance Prediction for Concurrent Software on Multicore Architectures. A Simulation-Based Approach.
ISBN 978-3-7315-1146-5

Band 33 **Manuel Gotin**
QoS-Based Optimization of Runtime Management of Sensing Cloud Applications.
ISBN 978-3-7315-1147-2

Band 34 **Heiko Klare**
Building Transformation Networks for Consistent Evolution of Interrelated Models.
ISBN 978-3-7315-1132-8

Band 35 **Roman Pilipchuk**
Architectural Alignment of Access Control Requirements Extracted from Business Processes.
ISBN 978-3-7315-1212-7

Band 36 **Stephan Seifermann**
Architectural Data Flow Analysis for Detecting Violations of Confidentiality Requirements.
ISBN 978-3-7315-1246-2

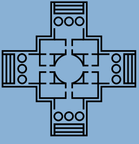
Band 37 **Sofia Ananieva**
Consistent View-Based Management of Variability in Space and Time.
ISBN 978-3-7315-1241-7

Band 38 **Robert Heinrich**
Architecture-based Evolution of Dependable Software-intensive Systems.
ISBN 978-3-7315-1294-3

Band 39 **Max Scheerer**
Evaluating Architectural Safeguards for Uncertain AI Black-Box Components.
ISBN 978-3-7315-1320-9

Band 40 **Sandro Giovanni Koch**
A Reference Structure for Modular Model-based Analyses.
ISBN 978-3-7315-1341-4

Band 41 **Maximilian Walter**
Context-based Access Control and Attack Modelling and Analysis
ISBN 978-3-7315-1362-9



The Karlsruhe Series on Software Design and Quality

Edited by Prof. Dr. Ralf Reussner

This work develops architectural security analyses to detect access violations and attack paths in software architectures. This is crucial due to the increasing digitalization. Access control policies, influenced by contextual factors, and vulnerabilities pose challenges, often analyzed separately. Existing methods lack integration and overlook system deployment nuances. The proposed solution employs software architecture models for early security assessment, promoting "Security by Design." Specific contributions include metamodels for access control and vulnerabilities, scenario-based access control analysis, and two attack analyses. Evaluation shows high accuracy in identifying access violations and compromised elements, with potential effort reduction. This approach aids in designing secure systems by estimating the impact of access control policies and vulnerabilities during software development.

ISSN 1867-0067
ISBN 978-3-7315-1362-9

Gedruckt auf FSC-zertifiziertem Papier