Automatic Detection of Architectural Security Flaws

Dipl.-Inf. Bernhard J. Berger

0000-0001-6093-9229

1^{st} reviewer:	Prof. Dr. rer. nat. Rainer Koschke
2^{nd} reviewer:	Prof. Eric Bodden, Ph.D
Defended:	07th March 2022

A dissertation submitted to
The faculty of Mathematics and Computer Science
At the University of Bremen

In Partial Fulfillment of the Requirements for the
Academic Degree of

Doktor der Naturwissenschaften (Dr. rer. nat.)

November 2021

To whom it may concern ...

Abstract

Software systems are increasingly interconnected, and more and more devices have a permanent connection to the worldwide web. While this is convenient for end-users and desired by companies whose revenue is increased through more information on their customers, it results in an attractive attack vector not only for criminals trying to scam people but also for industrial espionage. Therefore, software security and data privacy are essential topics nowadays. While there is plenty of research in the area of implementation-level security flaws, up until now, research has been lacking ideas to automate architectural security flaw detection.

This thesis presents the ArchSec approach, which investigates how architectural security flaws can be detected automatically. ArchSec uses static analyses to extract extended dataflow diagrams. An extended dataflow diagram is an extension to traditional dataflow diagrams adding types and attributes to them. The extracted diagrams are then converted into property graphs and checked for architectural security flaws using a knowledge base containing security anti-patterns and security patterns. Parts of the property graph matching an anti-pattern correspond to potential security flaws if no matching security pattern is found to mitigate it. Therefore, the detection process reduces to a subgraph isomorphism problem.

Several case studies demonstrate the feasibility of the ArchSec approach. Each case study shows transferable knowledge base rules and describes the found security flaws and applied mitigations in the context of the case study's applications. The case study applications contain Android apps, hybrid Android apps, and JavaEE applications. ArchSec identified different security flaws, such as authentication problems, authorisation problems, and cryptography-related problems. Consequently, it detected violations of most information security's protection goals. The detected flaws in real-world applications would have allowed attackers to spy on transmitted and processed data, circumvent existing authorisation constraints, and execute attacker-written code.

Combining a predefined knowledge base and an automatic extraction and detection process simplifies the threat modelling approach significantly and makes it feasible even for small and midsized companies. Decreasing the necessary effort for a professional architectural security assessment will hopefully result in

more secure software systems and improve the general state of data protection.

Zusammenfassung

Softwaresysteme sind immer stärker untereinander verbunden und immer mehr Geräte haben eine permanente Verbindung zum Internet. Während dies für Endanwender bequem ist und von Unternehmen gewünscht wird, deren Umsatz durch mehr Informationen über ihre Kunden gesteigert wird, ergibt sich daraus ein attraktiver Angriffsvektor nicht nur für Kriminelle, sondern auch für Wirtschaftsspionage. Softwaresicherheit und Datenschutz sind daher heutzutage wichtige Themen. Während es im Bereich der Sicherheitslücken auf Implementierungsebene eine Vielzahl von Forschung gibt, fehlten der Forschung bisher Ideen zur Automatisierung der Erkennung von Sicherheitslücken in der Architektur.

Diese Arbeit stellt den ArchSec-Ansatz vor, der untersucht, wie architekturelle Sicherheitslücken automatisch erkannt werden können. ArchSec verwendet statische Analysen, um erweiterte Datenflussdiagramme zu extrahieren. Ein erweitertes Datenflussdiagramm ist eine Erweiterung zu herkömmlichen Datenflussdiagrammen, die Typen und Attribute hinzufügt. Die extrahierten Diagramme werden dann in Eigenschaftsgraphen umgewandelt und mithilfe einer Wissensdatenbank mit Sicherheits-Antimustern und Sicherheitsmustern auf architekturelle Sicherheitslücken überprüft. Teile des Eigenschaftsdiagramms, die mit einem Antimuster übereinstimmen, entsprechen potentiellen Sicherheitslücken, wenn kein passendes Sicherheitsmuster gefunden wird, um es zu mindern. Daher reduziert sich der Detektionsprozess auf ein Teilgraphen-Isomorphismusproblem.

Mehrere Fallstudien belegen die Machbarkeit des ArchSec-Ansatzes. Jede Fallstudie zeigt übertragbare Regeln und beschreibt die gefundenen Sicherheitslücken und angewandten Gegenmaßnahmen im Kontext der Anwendungen der Fallstudie. Die Fallstudienanwendungen enthalten Android-Apps, hybride Android-Apps und JavaEE-Anwendungen. ArchSec identifizierte verschiedene Sicherheitslücken wie Authentifizierungsprobleme, Autorisierungsprobleme und kryptografische Probleme. Folglich erkannte es Verstöße der meisten Schutzziele in der Informationssicherheit. Die erkannten Fehler in realen Anwendungen hätten es Angreifern ermöglicht, übertragene und verarbeitete Daten auszuspähen, bestehende Autorisierungsbeschränkungen zu umgehen und von Angreifern geschriebenen Code auszuführen.

Die Kombination einer vordefinierten Wissensbasis und eines automatischen Extraktions- und Erkennungsprozesses vereinfacht den Ansatz der Bedrohungsmodellierung erheblich und macht ihn auch für kleine und mittelständische Unternehmen praktikabel. Die Reduzierung des notwendigen Aufwands für eine professionelle Sicherheitsbewertung der Architektur führt hoffentlich zu sichereren Softwaresystemen und verbessert den allgemeinen Stand des Datenschutzes.

Preface

Plenty of great research gives the foundation of this dissertation. Due to a large number of authors and publication channels, the publications are written in both British and American English. This doctoral thesis is written in British English, which raises the question of what happens to the American spelling of words. The thesis' author decided to adopt such terms for this written report in the interests of a uniform language image. As a result, terms such as Microsoft's Threat Modelling are not spelt with a single l, as is usual in the American original, but with double l. Otherwise, the flow of reading would have been disturbed by a constant change of spelling.

Another necessary linguistic decision was made regarding the spelling of the compound word *dataflow* that occurs regularly within this thesis, such as in *dataflow diagram*. Especially, the word *dataflow diagram* can be found in literature in three different ways of writing: *data flow diagram*, *data-flow diagram*, and *dataflow diagram*. This thesis uses the spelling suggested by the *Dictionary of Computer Science* from the *Oxford University Press*, which spells it *dataflow diagram* [49]. Consequently, in order to be consistent, the term *data flow* is written as *dataflow*.

Contents

List of Figures

List of Tables

List of Listings

Part I

Prelude

Chapter 1

Introduction

Nowadays, software security and data privacy are vibrant and significant topics [42, 72, 151]. Three decades ago, most computer and software systems were running independently and had no permanent connection to the world wide web or other large networks. Back in those times, the biggest security threat were other users or viruses transmitted via external data storage devices. Today, smartphones with mobile data connections are widespread, making us available all of the time. Our homes are connected to the world wide web most time of the day, too. We make our homes smart by installing sensors and actuators with network access, so that we can access and control them from wherever we are. We even connect ourselves with the help of fitness trackers that send health data to the cloud. One of the main objectives of current and emerging technologies, such as the Internet of Things, Cyber-Physical Systems, and Industry 4.0, is to connect an ever-growing number of systems [156].

Many of these systems handle personal and secret data that are of interest to other parties. Cyber-criminals spy on account details and credentials. Advertisers create user profiles as accurately as possible in order to provide location-based contextual advertising. Prosecutors utilise the usage and communication data for their investigations. Intelligence services predict threats to national security based on extensive data. Terrorists may create confusion and disrupt public peac by sabotaging critical infrastructures. These are just some reasons for the growing interest in and the dire need for software security and privacy of end-users and vendors. Beyond that, various reasons incentivise software vendors to handle the security aspects of their products with care. Surveys indicate that security breaches can have a negative impact on sales [142]. Some service providers have contractual penalties within their contracts, leading to severe financial strains when breached.

There are different approaches in industry and academia to secure an application. Static analysis tools that find security bugs at the implementation level have existed for several years and find different types of security issues, ranging from missing input validation (buffer overflows, SQL injection, path traversal)

to wrong API usage. These tools employ static type information and control and data flow analysis to identify potential security bugs. Nevertheless, these tools share a common problem since their produced results contain a large number of false positives. Some former research prototypes evolved to commercial products which are used by industry in their quality assurance process.

Experts claim that implementation-level bugs only make out half the security issues that are compromising a software system [115]. Therefore, the industry realised the need to ensure the application's security during its complete development lifecycle, resulting in efforts to define security-aware development processes. Microsoft, for instance, introduced the Security Development Lifecycle [84]. It consists of secure software design, continues with a security-conscious implementation and ends with security tests. An essential part of this approach is a manual examination of the software architecture by architects and security experts. Possible security flaws form a set of security requirements and their corresponding countermeasures. This architectural risk analysis or threat modelling helps to identify security flaws.

Academia, in turn, delves into language-based security as well. On the one hand, existing programming languages are extended by security information, such as the confidentiality of variables, to improve the aforementioned static security analysis. On the other hand, they add security annotations to source code which then can be used to prove security policies. Furthermore, academia develops approaches to creating formal models of software systems or (software) protocols and formally verifying them. The proven security properties differ. They range from authentication properties to authorisation properties or consistency rules within authorisation constraints. Unfortunately, verification approaches scale badly and are not able to handle industry-sized systems [29].

Although security experts agree that there is no perfect security, the need for securing software systems is indisputable. An extension to creating secure systems is to conduct risk management to security since the goal has to be to build software systems with good enough security [85, 138, 156].

1.1. Problem Statement

In an ideal world, software architects and developers consider a software system's security aspects from the first day on. This design principle is known as *Security by Design*. However, we do not live in an ideal world, and security is not a first-class citizen in software development. This results in subsequently installed security measures or applications secured by external measures, such as application firewalls. Neglecting investment in security in the first place

may have several reasons. For example, running the software system on the world wide web was not considered initially [14]. Another factor is the increased development time and cost while having no direct functional value for the customer [156]. When a developer finally decides to make a system more secure, the first step is to employ one of the easily accessible static-analysis tools mentioned above to find security bugs at the implementation level and fix them. However, fixing these security bugs covers only half of the security issues. McGraw states that half of the security issues are architectural-level security flaws [115]. These flaws are conceptual and fundamentally undermine an application's security, leading to situations where an attacker can bypass security measures as if there were any in place at all. Research gives evidence that fixing bugs is cheaper the sooner it is done in the application's lifecycle [11, 43, 96]. Therefore, fixing these flaws before the software's implementation is desirable.

Identifying security flaws at the architectural level is of crucial importance for the security of software systems. Proving that a system is secure is the ideal goal, but these approaches are too sophisticated for real industrial systems since the memory consumption and runtime are very high. Furthermore, the software has to exactly implement the model to ensure that the proven security properties hold for the implementation.

Large software vendors, such as Microsoft and SAP, use architectural risk analysis to identify potential security risks. Nevertheless, security experts in large companies have to support many applications, making it impossible to analyse these applications regularly. Small or medium-sized companies employ this approach very rarely. One reason for this is the lack of available experts within the company. Manual security analysis—especially at the architectural level—requires thorough knowledge of security concepts and their correct application. Therefore, most of the time, small or medium-sized companies hire external security experts to perform an architectural risk analysis. Even in large companies, security is not the task of developers all of the time. It is not unusual to outsource this task to a security team that is not part of the regular development team [132].

External security experts and internal security experts in large companies face a common problem. They are not familiar with the analysed software system. Therefore, an architectural risk analysis is an interview-like process. Security experts, software architects, and software developers participate in this discussion. The architects and developers describe the software under investigation in detail, beginning with the purpose of the software, its operating environment, the input and output, connections to other systems, and the frameworks used. From the gathered information, the security expert will then create an informal model and use it to discuss security aspects, such as the

data that are processed by specific components. Commonly, the model will be refined based on the discussions.

The security expert then takes the models to identify potential security flaws. There are some structured methods, such as STRIDE, an attacker-oriented approach to derive possible flaws. The result of this step highly depends on the knowledge of the security expert who is conducting the method. Naturally, even security experts do not know all possible fields of software security in depth. The resulting flaws are then part of a risk analysis where the expert estimates how difficult it is to exploit the flaws and how severe the impact will be. Based on the estimated values, the importance of the flaws is calculated, and the flaws are ranked accordingly.

However, this approach has several drawbacks. First of all, it is very time-consuming [62] because several people are involved in this process, and the discussion takes much time. For companies, the additional time results in additional costs they have to take into account [97]. Khala, Vanciu, and Abi-Antoun state that "architectural risk analysis, however, is much less mature and lacks compelling tool support" [98]. Moreover, the created models are inaccurate because they illustrate the application as developers see them, and there often is a mismatch between the implementation and the *mental* architecture [62]. Abi-Antoun and Barnes state: "One potential hurdle to achieving significantly better defect reduction is that the architecture may not show all the communication that is present in the system. As a result, an architecture-level analysis may be incomplete" [8]. Moreover, the method is very dependent on the security expert because she centralises all the knowledge within the process.

It is, therefore, crucial to not analyse the planned but the implemented architecture. To analyse the implemented architecture, it is necessary to extract an architectural representation that contains the relevant security information. The current architectural representations used in architectural risk analysis are not equipped to capture security information adequately. Therefore, the first research area this thesis is concerned with is developing more expressive dataflow diagrams. The second research area is the automatic extraction of architectural security views enabling a flaw detection process that works on the implemented architecture and not the planned one. As mentioned above, the results of this flaw detection process are highly dependent on the conducting security expert. Additionally, not every software developing company may have the financial means to engage a security expert in the first place. Consequently, the third research area is the automatic detection of security flaws. Table 1.1 summarises the three presented research areas for which Chapter 5 will develop six research questions, two for each area.

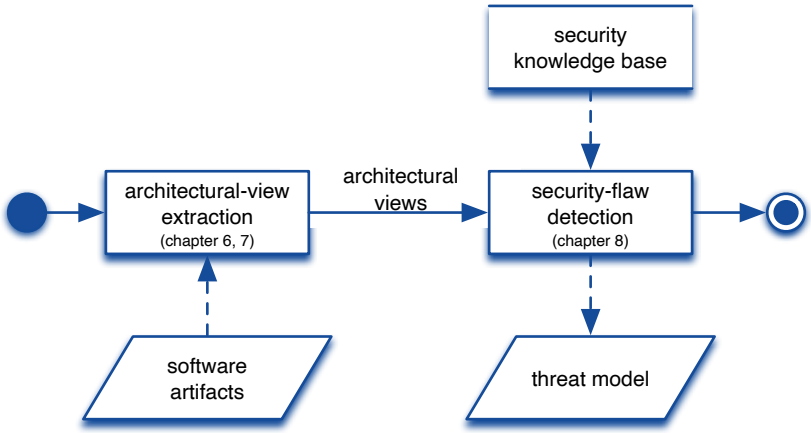
Table 1.1. Summarised research areas

Name	Topic
T1	*More expressive dataflow diagrams.*
T2	*Automatic extraction of architectural security views.*
T3	*Automatic detection of security flaws.*

1.2. Approach

As already mentioned, architectural risk analysis is an essential step in securing an application. For various reasons, it is, unfortunately, only seldomly performed. This thesis develops the ArchSec approach[1]—ArCHitectural SECurity—as a significant step towards automating the process of architectural risk analysis. In principle, ArchSec consists of two steps—depicted in Figure 1.1—and is aided with a knowledge base.

Figure 1.1. Process overview of the ArchSec approach



The first step uses static analysis to extract a system model from the software's implementation. Subsequently, it recovers an architecture view based on these models. The second step uses this view alongside a knowledge base of security flaws to automatically detect flaws for the analysed software. The result of this process is a threat model containing possible flaws and their preliminary risk estimation.

This approach improves architectural risk analysis in different ways. First

[1] ArchSec can be found online at https://www.archsec.de.

of all, automatically extracting security-related architecture views reduces the time necessary to conduct the analysis. Second, there is no difference between the analysed and the implemented architecture. The security analysis will be conducted based on the real implementation, thus improving the results. Third, the experts will require less technical knowledge since the extracted architecture is independent of technology. Finally, it is possible to automatically discover general architectural security flaws by reusing formalised flaw descriptions. New flaw patterns would be checked automatically for existing software systems.

Security experts can use the extracted architectural view to understand the security concepts of unknown software systems more quickly. This software comprehension task has, for instance, to be done by security evaluators in security-related certification processes, such as Common Criteria. The architectural view is not the only aspect a security expert needs to understand the security aspects of a software system correctly.

The explicit knowledge base supports different stakeholders in this process. As depicted in Figure 1.2, domain experts, technical framework experts, and security experts can externalise their security knowledge into a central security knowledge base. The automatic security-flaw detection uses this formalised knowledge to create a threat model for a given software artefact. The resulting threat model can be inspected by software architects, IT, and quality assurance, identifying necessary mitigations.

All the provided information may allow developers, software architects, quality assurance, and security experts to create more secure software systems.

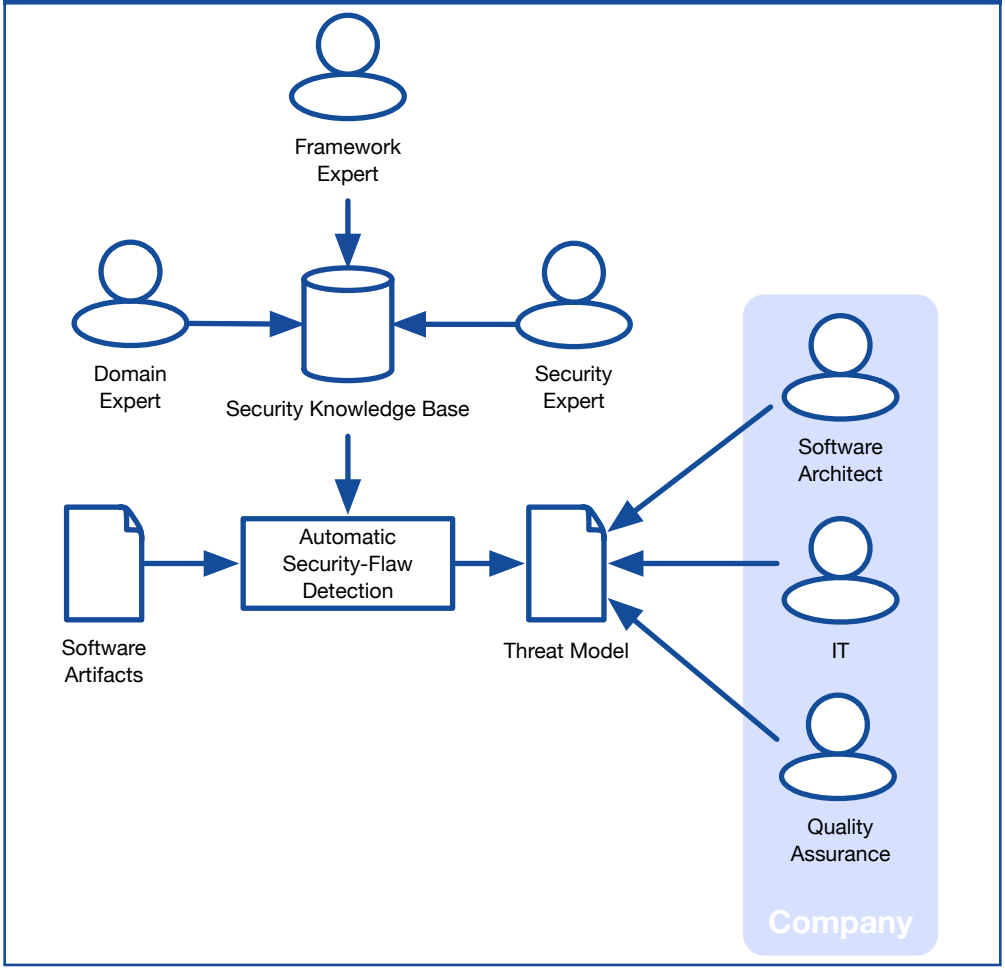
1.3. Contributions

This thesis presents four main contributions that lay along the path of the ArchSec approach.

Extended Dataflow Diagrams (EDFD): The informal definition of dataflow diagrams does not allow automatic detection of security flaws. Therefore, it is necessary to improve the semantics of dataflow diagrams. Chapter 7 presents a possible approach to do so in introducing extended dataflow diagrams. The more formal expressiveness of these diagrams is shown by successfully identifying security flaws using ArchSec, c.f. evaluation Chapters 10 and 11. This contribution, therefore, aims at research area *T1*.

Static EDFD extraction concept: The lack of existing architectural descriptions results in high expenses for the extended dataflow diagram creation. Consequently, Chapter 6 introduces an approach to statically extract EDFDs based

Figure 1.2. Stakeholder of the approach



on model-driven software development techniques. As of now, it supports Java Enterprise systems, Android applications, and hybrid Android applications consisting of Java, HTML and JavaScript code. The evaluations in Chapters 10 and 11 serve as support pillars for the applicability of this approach once again. Thereby, this contribution is in the range of research area *T2*.

Automatic Detection of Architectural Security Flaws: A manual detection of security flaws is time-consuming, error-prone, and subjective. As a solution to these problems, Chapter 8 proposes a mechanism to automatically identify architectural security flaws in extended dataflow diagrams. For this purpose,

ArchSec uses graph matching to identify insecure parts in EDFDs and potentially matching mitigations that secure the identified insecure parts. Several security assessments have been conducted using this technique, c.f. evaluation Chapters 10 and 11, relating to research area *T3*.

Knowledge Base of Architectural Security Flaws: As it turns out, there is a set of general and framework-specific security flaws. Chapter 8.2 introduces a knowledge base to capture common flaw descriptions that can be checked against every analysed system's extended dataflow diagram representation. Chapters 10 and 11 give several examples of existing knowledge base rules used in the case studies. This contribution adds to research area *T3*.

1.4. Context

Research of five different projects culminated in this thesis. Two of them have been funded by the German Federal Ministry of Education and Research, one has been funded by the German Federal Ministry for Economic Affairs and Energy, one is founded by the German research foundation, and the last one is founded by the Federal Ministry of Transport and Digital Infrastructure.

The project *ASKS – Architekturbasierte Sicherheitsanalyse geschäftskritischer Informationstechnik* started in 2010 and lasted for two years.[2] The goal of ASKS was to find architectural security flaws in business applications implemented using the Java Platform, Enterprise Edition. The project used the Bauhaus tool suite—a cooperate project of the universities Stuttgart and Bremen. Furthermore, there is a commercial spin-off called Axivion. The second project lasted for two years as well and is called *ZertApps — Zertifizierte Sicherheit für mobile Anwendungen*.[3] ZertApps started in 2014 and focused on using static analysis during the certification process of mobile applications. The project employs Soot—a static analyser framework for Java and Android. The third project is called *CertifiedApplications*[4]. The project extends the ideas of the *ZertApps* project. The goal is an extensible certification scheme that is elaborated for Android and Java Enterprise applications. Static analyses are an essential part of this approach that allows to speed up the certification process to reduce the required time and costs. The fourth project, *SecPatterns*, aims to understand the use and spread of security patterns. Security patterns are an analogous concept to design patterns known from the field of software engineering. Be-

[2]The grant number of ASKS is 01IS10015B.
[3]ZertApps was funded by the BMBF under the number 16KIS0074.
[4]The BMWi fund number of CertifiedApplications is ZF4122102.

sides the mentioned goals, *SecPatterns* also aimed at automatically identifying security patterns' implementations in existing applications.

The last project, named *SecProPort* and its predecessor *PortSec* dealt with the security in port systems. These systems are characterised by a large number of interfaces and connected software systems. In the context of these projects, different evaluations have been conducted.

1.5. Previously Published Content

Several parts of the ArchSec approach have already been presented at different international conferences. Table 1.2 shows an overview of these publications. Additionally, Table 1.3 gives an overview of additional publications related to the thesis topic but are not part of the thesis itself.

The first paper related to this thesis is an idea paper published at ESSoS 2010 [147]. The evaluation of this publication used a modified version of Bauhaus, a reverse engineering tool suite, to extract the enforced access policy of a JavaEE example application and compared it, using the reflection analysis, to the intended access policy. The approach uncovered a difference between the planned and the implemented authorisation policy, even for the analysed tutorial application published by the originators of Java enterprise. The first paper author developed the underlying idea, while the thesis' author implemented and evaluated it.

The following paper was published in the year 2011. It reported on using Bauhaus to understand the enforced access control policy of the Bluetooth component within the Android framework. We used Bauhaus to identify the concrete enforcements, the enforced permissions and their relation towards each other. Furthermore, the paper inspected a security pattern that is used within the Android framework [31]. The thesis' author extracted views of the Bluetooth component and inspected the enforced permissions, whereas another author inspected the security pattern.

An approach to extracting architectural information on specific kinds of Java enterprise systems and identifying information flows between different users in Java-based web applications was published at IFIPSec 2012. In its essence, this publication identifies the wrong usage of the Object Pooling Pattern that can lead to information flows between users [35]. The thesis' author found the problem in the context of the ASKS project, developed the analysis, and evaluated it.

In 2013, a first publication described an architectural analysis that allows tracking confidential platform data, such as location information, over multiple

Table 1.2. Previous publications of thesis' content

Reference	*Title*	*published at*	*Chapters*
[147]	Idea: Towards Architecture-Centric Security Analysis of Software	ESSoS 2010	10.1
[31]	An Android Security Case Study with Bauhaus	WCRE 2011	11.1
[35]	An Approach to Detecting Inter-Session Data Flow Induced by Object Pooling	IFIPSec 2012	10.2
[24]	The Transitivity-of-Trust Problem in Android Application Interaction	ARES 2013	8, 11.2
[37]	Extracting and Analyzing the Implemented Security Architecture of Business Applications	CSMR 2013	7, 8, 10.3
[36]	Architekturelle Sicherheitsanalyse für Android	D • A • CH Security 2014	7, 6.3, 8, 11.3
[107]	Taint Analysis of Manual Service Compositions using Cross-Application Call Graphs	Saner 2015	6.3
[114]	Zertifizierte Apps	BSI Kongress 2015.	7
[38]	Automatically Extracting Threats from Extended Data Flow Diagrams	ESSoS 2016	7, 8, 12.1
[39]	The Architectural Security Tool Suite – ArchSec	SCAM 2019	7, 8
[34]	Static Extraction of Enforced Authorization Policies - SeeAuthZ	SCAM 2020	3.4.2, 6.3
[33]	Threat Modeling Knowledge for the Maritime Community	MareSec 2021	12.2

Android components and apps. Thus, allowing for tracking data passed with the help of inter-process communication means and sent to an external receiver [24]. The thesis author's contributions were conceptual ideas for the analysis approach, their realisation, and support for the case study. In the same year, a published paper at CSMR reported on extracting the implemented security architecture for Java enterprise systems. This paper presented the first parts of the final ArchSec approach to extract architectural views for a given software system using reverse-engineering techniques. The architectural views were then used to identify security flaws with the help of OCL invariants [37]. The thesis' author was mainly responsible for the idea and contents of the paper.

In 2014, the thesis' author showed at D-A-CH Security that the ArchSec approach—published at CSMR'13—is transferable to hybrid Android Apps, which contain Java code, as well as HTML and Javascript. The paper showed that several Apps use bare HTTP connections to load remote content, allowing hackers to make a man-in-the-middle attack [36]. In the very same year, a second paper was published that describes how static analysis and automated architectural risk management can be used for certifying Android Apps [23]. The thesis' author contributed information on his research. The contents of this paper are not part of this thesis.

At Saner 2015, the author published a paper on using cross-application call graphs to get better taint analysis results for web service-based software systems [107]. The thesis' author provided ideas on analysing Java enterprise systems and the work built on top of ArchSec's analysis pipeline to create entry points for Java enterprise systems for connecting the call graphs of client and server application. In the same year, the thesis' author co-published a paper at a conference of the german federal office for information security detailing the process of certifying Android apps. This publication is a following up paper to the paper from 2014. The paper shows how architectural security views and flaw detection can help security auditors in their daily routine [114].

In 2016, the thesis' author published a paper at Essos 2016 on automatically extracting security threats from extended dataflow diagrams. In contrast to the paper published in 2013, it focuses on identifying potential security flaws in manually created architectural views. A second difference to the previously published papers is using a graph-based query language to describe the security anti-patterns and mitigations [38]. The thesis author was responsible for the paper's concept, its execution, and the writing.

In 2019 the thesis author published a paper on the complete architectural security flaw extraction and detection environment—ArchSec. The paper describes the extraction and checking of extended dataflow diagrams in greater detail than the previously published papers and lists different use cases for

the environment [39]. The thesis' author was the main responsible for this paper. Additionally, a paper published in 2019 reports on a static analysis based approach to verify the implementation of multi-model access control properties [32]. This paper is not part of this thesis but is related to the authorisation pattern investigated in this paper. The thesis author conceptualised the underlying model, wrote the static analysis to extract the model information, and conducted one of the presented case studies.

The contribution to the SCAM'20 presented the static extraction of enforced authorisation policies. Therefore, the approach uses inter-procedural analyses to determine which resources are protected using which permissions. To this end, it introduced the authorisation pattern, which is also presented in this thesis. The thesis' author was the main responsible for this paper.

Table 1.3. Further papers published by the thesis' author in the thesis' topic

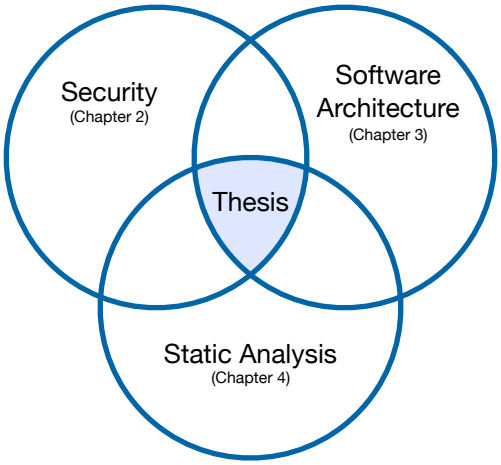
Reference	*Title*	*published at*
[23]	Zertifizierte Datensicherheit für Android-Anwendungen auf Basis statischer Programmanalysen	GI Sicherheit 2014
[32]	Towards Effective Verification of Multi-Model Access Control Properties	SACMAT 2019
[118]	eNYPD — Entry Points Detector – Jakarta Server Faces Use Case	SCAM 2021

Finally, in 2021 a paper on automatically generating security flaw rules was presented at MareSec to automatically check BPMN-based extended dataflow diagrams. This approach allows non-security experts to specify an authorisation policy using a simple matrix-like editor instead of writing graph-query rules. The thesis author was the main responsible for this paper. Additionally, a paper was published that presents eNYPD. eNYPD is a tool for detecting entry points in Java enterprise applications. An entry point is a method that can be called externally and, therefore, might be used by an attacker to attack a system.

1.6. Thesis Outline

This document divides into four parts. First, **Part I**—the Prelude—gives an overview of this thesis' topic and the foundations of the thesis' content. Here, Chapter 1 presents a problem statement and briefly explains the thesis' proposed solution. Since the thesis deals with different topics, shown in Figure 1.3, this part summarises the knowledge used throughout the remainder of the document. Chapter 2 introduces a broad range of security topics necessary to help classify these topics in the remainder of the document. Chapter 3 introduces software architectures, how they may be created and documented, and sheds light on how the architecture can support the security of a software system. Finally, Chapter 4 gives some information on static-analysis techniques helpful in extracting architectural information to automatically re-document software systems and automatically detect security flaws.

Figure 1.3. Thesis relation to research topics



Second, **Part II**—the Theme—describes ArchSec, an approach for automated threat modelling and an architectural risk analysis environment, developed for this thesis. ArchSec automatically extracts software architectures for component-based systems and automates the detection of security flaws in software architectures. Starting this Part, Chapter 5 elaborates the research topics and poses research questions this thesis answers. Afterwards, Chapter 6 introduces the static analyses framework part of ArchSec used to extract architectural information and security information. Next, Chapter 8 introduces extended dataflow diagrams, a more expressive version of dataflow diagrams, used in this thesis. Finally, Chapter 8 presents a description language

for potential security flaws and explains how the matching process works.

Third, **Part III**—the VARIATIONS—presents different case studies showing that the ARCHSEC approach applies to various kinds of software systems and security flaws. Therefore, Chapter 9 derives research hypotheses for the remaining—since not yet answered—research questions. The following case studies collect evidence for these hypotheses. Chapter 10 report on several case studies for Java enterprise software using automatically extracted software architectures. Additionally, Chapter 11 investigates a larger number of Android apps. Chapter 12 focuses on ARCHSEC's detection mechanism by reporting on findings on manually created software architectures. All three chapters report on different security flaws uncovered by the automatic detection process.

Finally, **Part IV**—the FINALE—gives an overview of the research related to the thesis' topic in Chapter 13 to place this work in the context of existing research. Lastly, Chapter 14 concludes the thesis and critically discusses the results found in the light of the research hypotheses and research questions. Furthermore, it identifies open research questions to guide other researchers interested in automated security flaw detection.

1.7. A Hitchhiker's Guide to this Doctoral Thesis

Depending on the time the earth remains to exist, there are different ways to read this thesis.[5] If time is very limited, one can stop reading quite after this chapter. The introduction already mentioned the topics of the thesis. However, if there is time to go into more details, it is advisable to read at least Chapters 5, 9, and 14. These chapters develop the research questions, derive the research hypotheses for the evaluation and conclude the findings. This reading order skips all the nitty-gritty details and is the condensed reasoning thread of the thesis. If time is not a problem at all, the remaining chapters can be read depending on the reader's interests and knowledge. Chapters 2, 3, and 4 help to get the basic ideas of software security, software architecture, and static analysis to understand the details of the ARCHSEC approach presented in Chapters 6, 7, and 8. If it is of more interest what the ARCHSEC approach can do in reality, it is definitely a good idea to take a look at the case studies in Chapters 10, 11, and 12.

[5] In making a decision, please consider the earth's destruction to facilitate an intergalactic highway construction project for a hyperspace express route. And please remember to carry a towel.

Chapter 2

Software Security

Security is an extensive and complicated subject that cannot be described in total within this thesis. Thus, this chapter focuses on the topics relevant to the remainder of the thesis and shows possible research topics in this area. Consequently, it first introduces information security in Section 2.1 and software's properties that need protection. Section 2.2 gives a brief overview of access control concepts since the evaluation of this thesis' approach (Chapter 10–11) identifies different access control mechanisms. Finally, it provides some approaches to creating secure software by using secure development processes in Section 2.3.

2.1. Information Security

This section describes information security, according to Claudia Eckert's textbook on IT security [65]. Information security aims to protect sensitive information or objects processed by or stored in a software system. The objects containing the information and the information itself are called assets. Subjects, such as users or other programs, operate on those assets using the system's operations. Therefore, it is necessary to protect the operations working on sensitive information, as well. The interaction of a subject with an asset or an operation operating on an asset is called *access*. There are different kinds of accesses, such as *read access*, *write access*, and *operational access*. An access control policy is the collection of circumstances under which subjects can access assets of the system. While authorization deals with the question of whether a subject is allowed to access specific information, authentication tries to answer whether a subject is the subject it claims to be.

The topics of safety, protection, and privacy are tightly coupled with security. A system is safe if it works as planned at all cost. Safety aspects are crucial for programs, such as embedded systems in the medical section, that can harm human beings. For these systems, it is essential that they do not behave unexpectedly. In data protection, the goal is to avoid loss of information under

any circumstances. This task is challenging if data has to be protected in the case of the storing device's power loss or hardware failure. Finally, privacy deals with the possibility of human beings controlling the sharing of their personal information. Since the European Union issued the General Data Protection Regulations [1], privacy has become more relevant for companies offering services in Europe.

Every software system has its own security, privacy, and data protection policies. Even if two applications process the same information, they likely have different policies. A quick example: A photo library application on a smartphone and a social media application access the device's location. For a photo library application, it would be suspicious if it sends the information to the worldwide web, in contrast to the social media application where it is accepted.

The following subsections introduce the security properties that need to be protected to make a system secure and give a first idea of how these properties' security can be achieved.

2.1.1. Authenticity

In most public accessible software systems, it is important to know who a system's current user is. Authenticity describes the credibility with which an object or subject identifies itself to confirm that it is the object or subject that it claims to be. In the real world, we directly interact with other people and determine who we are talking to. If we do not know the people we are dealing with, we can identify them with an identity card's help. For this purpose, accounts are created in software systems that can be clearly identified via a user name or an e-mail address. A subject then identifies himself with a combination of an account ID and a secret. The secret can be a password, some biometric features or digital keys. The combination of account and secret is often referred to as credentials. Besides authenticating people, it might be necessary to authenticate other software or hardware systems. It is an easy problem to solve for a limited set of known subjects since administrators can directly exchange their digital identity cards. As soon as more and more subjects are part of this system and distributed worldwide, it is necessary to have some trustworthy central party who can create some identity card for other subjects. For instance, for trustworthy communication in the worldwide web, these are so-called root certificate authorities. These root certificate authorities are trusted to check the identity for certificates, a digital identity card, they issue.

2.1.2. Integrity

A system protects integrity if it ensures that no subject can manipulate sensitive information unauthorised and unattended. To preserve integrity, it is necessary to define the write access to sensitive information for existing subjects. Within a software system, it is crucial to enforce that only permitted subjects manipulate the sensitive data. Therefore, it has to guarantee the subject's authenticity. Other mechanisms have to be used in situations where manipulation cannot be ruled out, such as unsecured transmission or data storage. Digital signatures, for instance, use asymmetric cryptography to protect information's integrity and detect manipulations by others. A cryptographic hash is calculated of the data to be signed to create a digital signature. The calculated hash is then transformed using the subject's private key. Every other party can verify the data's integrity by using the same hash function on the received data and confirming the digital signature with the issuer's public key. If the two values match, the data integrity is guaranteed.

2.1.3. Confidentiality

If a system makes sure that no unauthorised subjects can extract sensitive information from the software system, it protects confidentiality. While integrity means that no unauthorised write access of information is allowed, confidentiality forbids unauthorised read access. Security research shows that there are different ways to read out sensitive information. First, it is possible that the system transfers data between distributed system components or that it stores information in files. In these cases, state-of-the-art encryption of the data during transmission or storage is sufficient protection. Second, a subject can legitimately read the information and pass it on to another subject who must not access it. This confinement problem often called confused deputy problem, is more complicated, almost impossible, to solve with pure software measures. Approaches are using operating system extensions to label data with additional confidentiality information. The operating system then enforces confidentiality even if an unauthorised subject can access the data's file. Lastly, an attacker can use interference to get access to confidential information. There are attacks on the hardware- or software-level that try to read out confidential information indirectly via time measurements, consumption measurements, or covert channels. Here, too, a software solution is hard to implement.

2.1.4. Availability

A system is available when no unauthorised subject can prevent an authorised subject from interacting with the system. A computer system has various finite resources in a computer system, such as computing time, main memory, disk space, or network bandwidth. A system that ensures availability must ensure that unauthorised requests do not use up any of the resources, but they remain available to the system's authorised users. Ensuring availability is possible, among other things, by using schedulers, resource quotas, or firewalls to make sure a system is available. However, this task is very challenging when assuming an attacker with unlimited resources. If several thousands of unauthorised systems simultaneously access a network-accessible system, existing countermeasures will likely fail. In such cases, load balancing techniques can help to distribute the system over several processing nodes.

2.1.5. Non-Repudiation

If a subject cannot deny the operations it performed, the system ensures non-repudiation. Software systems billing the services they offer, for instance, have to document the operations made. Another example is legally binding communication between lawyers and courts, such as the German "*Besonderes elektronisches Anwaltspostfach*". Depending on the requirements, there are different degrees to guarantee non-repudiation. While it is sufficient for an e-commerce application to use simple logging mechanisms, legal correspondence has stricter standards and uses digital signatures to ensure non-repudiation.

2.1.6. Anonymization and Pseudonymization

Since the advent of privacy, anonymisation and pseudonymisation are relevant protection goals in information security. Anonymisation ensures that it is not possible to relate personal information to the connected subject anymore. Anonymisation must be ensured, for example, in surveys to protect the respondents' anonymity. With pseudonymisation, personal information is assigned to a pseudonym. The data must not allow any conclusions to be drawn about the identity of the subject. The requirements of pseudonymisation are less strict. In pseudonymisation, only a limited and permitted group of subjects can identify the information's related subject. In this case, the subject's information is stored with a subject's identifier. The subjects' assignment to their identifier is stored in a separate location, allowing legitimate subjects to assign the subject to its information.

2.1.7. Summary

This section showed that there are very different points that are relevant to information security. As shown, cryptography can help to ensure some of these topics. For other problems, it is necessary to define an authorisation policy. It is possible to distinguish between different operation kinds, such as *read access*, *write access*, and *operational access*. In literature, there are different approaches defining access control policies with various advantages and disadvantages. The following section documents some of the existing access control models in greater detail.

2.2. Access Control

The previous section described the properties that information security tries to protect. As illustrated, it is necessary to have an access control list to protect integrity and authenticity adequately.

Access control deals with the question in what way and under which circumstances a subject is allowed to access specific objects (or assets). The tuple of operation and object is called permission. Permissions are assigned to subjects in different ways that depend on the used access control model. Depending on the access granularity, the possible permissions are fundamental, such as reading, writing, executing, or object-dependent, such as printing or sending. Access control models regulate access to objects, but they do not allow the flow of information between an object and a subject to be restricted. Therefore, they cannot fully protect the confidentiality of information. If adequately implemented, access control can prevent unauthorised access to information. Nevertheless, it is impossible to undercut the fact that a subject may pass on the information to unauthorised third parties.

The security research community has developed different approaches to defining access control. The remainder of this section gives an overview of some well-known and frequently used access control models. The following subsections are based on Eckert's textbook [65] as well.

2.2.1. Access Control Strategies

There are different categories of access control models. First, there is the mandatory access control model strategy or shortly MAC. In a mandatory access control model, an external instance, such as the operating system, constraints and enforces a subject's permissions. A subject has no means to change the permission assignment in any way. Discretionary access control models (DAC)

deal with situations where there are owner-centric objects. The owner of an object, e.g. a user, creates an object, for instance, a file, and can grant access to other users by specifying file access permissions. Finally, there are role-based access control models. A role-based access control model can either be mandatory or discretionary. In contrast to the introduced strategies, role-based access models assign permissions to roles instead of subjects. Subjects, in turn, can belong to different roles.

2.2.2. Access Matrix Model

Lampson's access matrix model is the simplest model for mapping access rights [105]. It compares existing objects and subjects in a table, and the cells contain the corresponding authorizations. An access matrix is not necessarily static. The size of the matrix can change over time due to the creation and deletion of new objects or the addition of new subjects.

Table 2.1. Fictitious access control matrix for a clinic

Object	**Subject**					
	Nurse 1	***Nurse 2***	***Nurse 3***	***Doctor 1***	***Doctor 2***	***Doctor 3***
clinical record 1	*read*		*read*	*read*, *write*		*read*, *write*, *relocate*
clinical record 2	*read*		*read*	*read*, *write*		*read*, *write*, *relocate*
clinical record 3	*read*	*read*			*read*, *write*	*read*, *write*, *relocate*
clinical record 4	*read*	*read*			*read*, *write*	*read*, *write*, *relocate*

Table 2.1 shows an exemplary access matrix for a clinic. The matrix contains six different subjects, and four distinguished objects. The permissions are `read`, `write`, and `relocate`. According to their department, `Nurse 1`, `Nurse 2`, and `Nurse 3` have permission to read the `clinical records`. Next, `Doctor 1` and

`Doctor 2` have permission to `read` and `write` their patients' clinical records. Finally, `Doctor 3` is allowed to read and write all the files. Furthermore, he is allowed to relocate the patient and the belonging clinical record to another department.

2.2.3. Bell-LaPadula Model

The Bell-LaPadula model is one of the oldest formally specified access control models in computer science. It is rooted in the US military and is a mandatory access control model. It consists of a fixed set of permissions and extends the access control matrix model with a set of ordered trust levels. The set of permissions are append, control, execute, read-only, and read-write. The append permission allows the subject to append data to an existing object. The control permission enables a subject to grant and delegate permissions. To execute an executable file, the execute permission is required. Next, the read-only permission allows reading an object, while the read-write permission allows one to write it.

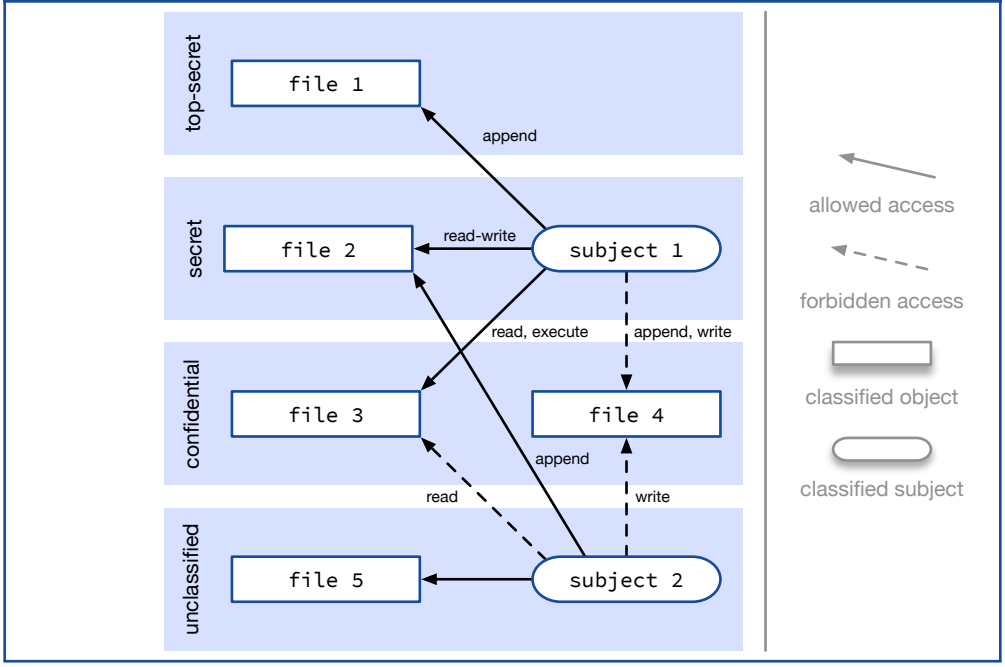
As already mentioned, the security classes are ordered and assigned to the subjects and objects. For objects, the assignment means that the data requires the appropriate level of protection. For a subject to access this object, it must have a clearance of the corresponding or higher security level. The Bell-LaPadula model ensures that a low clearance subject cannot read any objects with a high security requirement. Furthermore, it ensures that a high security clearance subject cannot write to objects with a low protection requirement.

Figure 2.1 shows the allowed and not allowed flows in the Bell-LaPadula model. In this example, there are four security classes: *unclassified*, *confidential*, *secret*, and *top-secret*. The graphic shows subjects as ovals and objects as rectangles. Simple arrows show which accesses are granted for which authorisation level. In contrast, dashed arrows depict prohibited flows. In summary, it can be said that appending data is allowed even with more strictly protected data, reading and writing at the same level and reading for the same and lower levels.

2.2.4. Role-Based Access Control Model

Ferraiolo and Kuhn introduced the role-based access control model (RBAC) in 1992 [71]. Sandhu extended it in 1998 [71, 139], which was the foundation for the ANSI/INCITS 359-2004 standard [17]. In addition to the concepts of subject and object, RBAC also introduces the indirection level role. A role is based on the task that a subject is currently processing with the system. Following the task, the subject requires permissions. These permissions are no

Figure 2.1. Allowed flows in the Bell-LaPadula model according to Eckert [65]



longer linked directly to the subject in the role-based authorisation model but to a role. A subject can be assigned to one or more roles to fulfil his tasks. This grouping of authorisations allows for simpler authorisation management since permissions no longer have to be selected manually for each subject. Instead, it has to be decided which tasks a subject can carry out with the system, and the corresponding roles are chosen.

Figure 2.2 shows the associations in the RBAC model according to the ANSI/INCITS 259-2004 standard. Users can have multiple roles. The user's session stores the currently active role for the given user at a given time. A user can have several active sessions simultaneously, where he carries out different subsets of his roles. Any number of permissions can be assigned to the roles. Permissions can either refer to the access of objects or the execution of operations. In addition to roles, the RBAC standard also introduces optional extensions. These extensions include hierarchical roles and validity conditions, such as the separation of duties. Separation of duty allows specifying that a subject is never allowed to be assigned to two specific roles at the same time.

The access model from Table 2.1 can also be represented using the RBAC

Figure 2.2. Associations in RBAC, according to ANSI/INCITS 359-2004 standard [17]

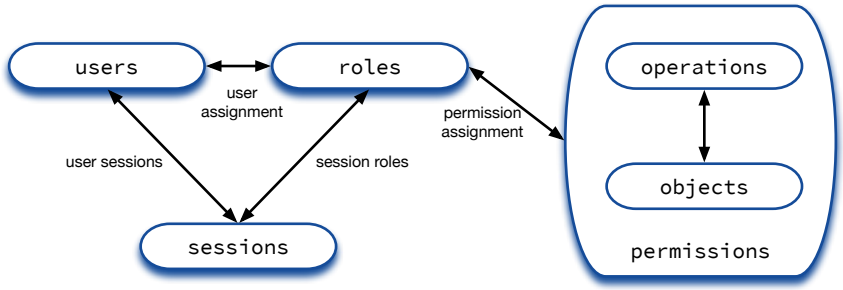
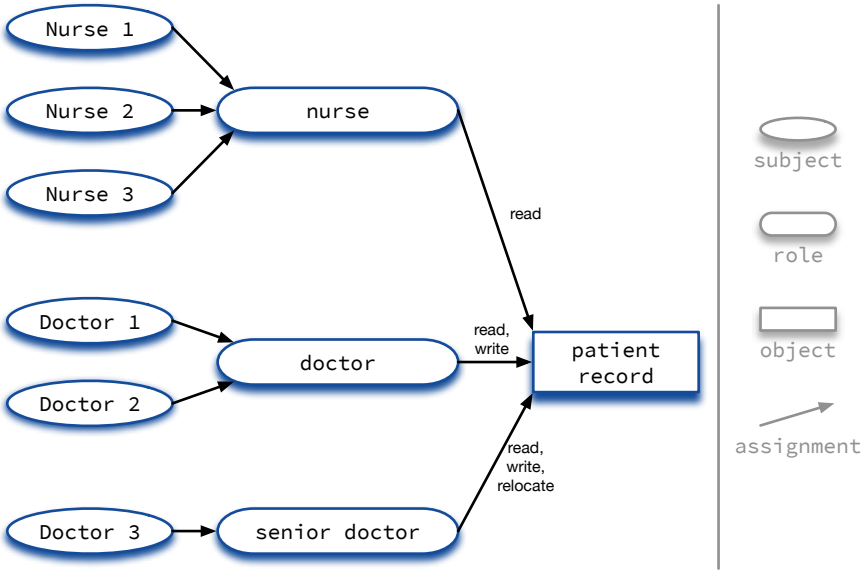


Figure 2.3. RBAC policy for the examplary access control policy shown in Table 2.1



model. Figure 2.3 shows an exemplary realisation. The left side of the figure contains all subjects from the access matrix. The subjects `Nurse 1`, `Nurse 2`, and `Nurse 3` are assigned to the role `nurse`. Correspondingly, `Doctor 1` and `Doctor 2` also have a role named `doctor`, whereas `Doctor 3` is assigned to the role `senior doctor`. The `nurse` role has `read` access to `patient records`. A `doctor` is allowed to `read` and `write`, and a `senior doctor` has the additional permission to `relocate` a patient (and the patient's record).

Please note that the shown RBAC model is not entirely equivalent to the access matrix shown. In this model, all nurses can read all patient's records, and all doctors can write them. The explanation of the access matrix model mentioned that these nurses and doctors belong to different departments. In a plain RBAC model, it is necessary to create other roles for each department, for instance, cardiology-nurse and cardiology-doctor. There are concepts, such as attribute-based access control, that overcome this shortcoming [103].

2.2.5. Summary

Access control is a fundamental matter in a multi-user system. This section introduced several concepts of access control. After a short explanation of the different access control strategies, it introduced three access control models. Each model has its advantages and shortcomings. While operating systems still work with adaptions of the access matrix, today's business applications frequently use RBAC-based models.

There are several techniques for securing software. The previous two sections introduced some of them. However, today's software still contains security flaws. For this reason, the following section examines the question of what a software development process must look like so that it promotes the development of secure software.

2.3. Securing the Software Development Process

Microsoft published its internal process for secure software development. Several companies adopted it, and Section 2.3.1 explains its ideas. Afterwards, Section 2.3.2 gives more details on their Threat Modelling approach, which they highly promote. Finally, Section 2.3.3 concludes this section.

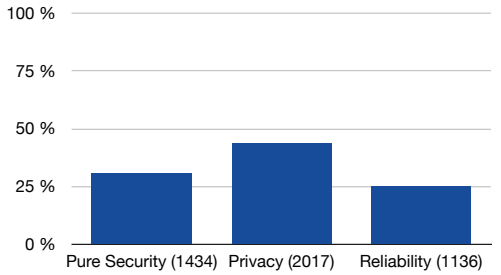
2.3.1. Microsoft's Security Development Lifecycle

Microsoft's Security Development Lifecycle (SDL) is Microsoft's answer to many security problems found in their operating systems in the late 1990s and early 2000s. In 2006 Howard and Lipner described Microsoft's approach to developing software more securely [83]. The rest of the section summarizes the contents of their book.

Howard and Lipner have determined that more and more different aspects of software security have become relevant nowadays. In addition to the classic security issues, privacy, and reliability developed to be increasingly important. They support their statement by categorising the Common Vulnerabilities and

Exposures entries created between 2002 and 2004. Figure 2.4 shows the results of this categorisation.

Figure 2.4. Distribution of CVE entries between 2002 and 2004 on the topics of security, privacy, and reliability [83]



31% of the entries deal with pure security issues, and 44% are related to privacy issues. The remaining 25% of the entries relate to reliability issues. According to the authors, Microsoft looked for a way to tackle all these issues. In total, Howard and Lipner describe 13 stages to make software development security-aware.

Education and Awareness The zeroth stage of Microsoft's Security Development Process consists of training everyone involved in the software development process to create awareness for secure software development. By not only training the developers but all participants of the software's lifecycle, such as managers, testers, and UI designers, security should also become a concern for everyone. The training should point out error patterns and teach how secure software is planned and programmed. The level of employees' training should be continuously measured and monitored, and they should be given incentives to deal with the issue of security.

Project Inception This stage initially introduces the Security Development Lifecycle and makes security a permanent software development aspect. First, the applications for which the SDL is relevant are identified. These are, for example, all applications that communicate over the Internet. For each identified project, a security advisor is appointed to keep an eye on the project's security aspects. For this purpose, bug tracking should also be adapted to track security and privacy problems explicitly. Accordingly, security issues in the development process are transparent.

Define and Follow Design Best Practices The second stage introduces common secure design principles in the software development process. These design

principles include accepted concepts such as least privilege, separation of privilege, and fail-safe defaults. Besides, an attack surface analysis for the planned software is carried out, in which it is identified how the software communicates with its environment. Then an attempt should be made to reduce the attack surface.

Product Risk Assessment The product risk assessment identifies the required security level for various parts of the project. It determines components modelled with the help of a threat model, components a security design review is carried out for, components examined using penetration testing, and components where fuzz testing is carried out for.

Risk Analysis The risk analysis systematically examines a software system for potential attack vectors. It is an attacker-centric approach, in which each component of the software is examined to see how an attacker could attempt to attack it. For each identified attack, an estimation of the likelihood and significance of the impact is conducted—furthermore, the risk analysis documents possible mitigations to the identified threats.

Creating Security Documents, Tools, and Best Practices for Customers This stage tries to improve the running applications' security through user documentation and additional configuration tools. The documentation explains to users and administrators how the application can be adequately secured. With the help of the additional configuration tools, securing an application should be made as simple as possible to ease beginners' hurdle. This idea follows the concept of fail-safe defaults mentioned in the stage Define and Follow Design Best Practices.

Secure Coding Policies The secure coding policies prevent developers from using functions, libraries or compiler options that make a program insecure. Besides, the authors recommend the use of static analysis tools to identify security problems automatically. The secure coding policy documents all chosen measures to make them replicable.

Secure Testing Policies While the previous stages deal with secure planning and implementation of software systems, stage seven takes care of security during the test phase. Microsoft's SDL uses fuzz testing, penetration testing, run-time verification, threat model review and attack surface reevaluation in this stage. Fuzz testing generates malformed input data to examine data parsers for reliability problems. Penetration testing examines a program's interface that can be reached via the network for possible weak points by executing known attack patterns. The run-time verification checks during run-time whether the

program behaves as specified or whether there are any deviations. Finally, both the threat models and the attack surface are updated to check the security of changes during implementation.

The Security Push The security push deals with old system parts developed before the SDL has been implemented. It is carried out when an application meets all the requirements for the next release. Its purpose is to take care of legacy issues existing in the software system. First, developer training takes place. Then, code reviews are conducted. The threat models are updated to reflect the legacy components' threats based on the code reviews' findings. Next, the legacy components are tested for security. Finally, the security documentation is updated if necessary.

The Final Security Review The final security review checks before the release whether the SDL process has been adhered to. Therefore, the final security review checks the results of the previous stages. Additionally, the threat models are reviewed, and all remaining security bugs are checked. Lastly, the final security review tests whether the release generating tools were configured correctly to guarantee a secure release.

Security Response Planning It is highly unlikely to create perfectly secure software. The security response planning, therefore, prepares the procedure when a security error is reported. The authors emphasise that it is not useful to fix security problems in an ad hoc manner. Instead, this stage plans a process for these situations. This process ensures that the fix works and the patch is built securely. It is also crucial that the security response team responsible for handling security bugs works with the development team to locate and fix the bug.

Product Release This stage is the regular product release without any additions.

Security Response Execution The final stage is the security response execution. The authors suggest taking all reported incidents seriously. They stress it is essential to follow the security response plan and not to be too quick.

2.3.2. Threat Modelling

In 2004, Swiderski and Snyder published a book on Microsoft's threat modelling approach [153]. To the authors, threat modelling is the logical continuation of their previously published security measures. They state that in the beginning, attempts were made to establish security with the help of network security

through firewalls since errors in the network stack were the most common gateways for attackers. When these measures were no longer sufficient, as attackers began to attack faulty software systems, attempts were made to make those more secure through security reviews. Specific security errors at the implementation level, such as buffer overflows, were identified and then rectified. Threat modelling now raises security to the architectural level and tries to identify architectural security problems to counter the next generation of problems. Swiderski and Snyder define the difference between them as follows:

> *An architecture issue can reasonably be remedied by secure design practices. The application's architects are responsible for such issues.*
>
> *An implementation issue is a problem that even the most junior developers would be expected to avoid, based on the team's secure-coding guidelines*

The steps in software development are constructive in nature. They deal with the software's correct implementation of functional requirements. Here, threat modelling takes on a special role since it primarily takes the attacker's perspective and identifies potential attack vectors. Therefore, the threat modelling process proceeds as follows. First, an architectural model of the software is created. This model contains the software's *assets*, *entry points*, *external dependencies*, and *interactions/dataflows*.

An asset is a system's resource that needs protection (please compare to the term asset from section 2.1)). A threat model contains all assets of a system. An entry point is a system's interface that allows interaction with the system from the outside. The interface can be any processed data (dataflow) or a callable interface (control flow). Processes are components of the system, which can be decomposed hierarchically. External entities are external components used by the software system to do its task. Finally, interactions and dataflows show the system's internal flow of data and control.

According to Swidersky and Snyder, a *threat* is *the adversary's goal, or what an adversary might try to do to a system*. The collection of all identified threats of a system is the *threat profile*. Furthermore, the authors suggest categorising external entities, entry points, and assets in different *trust-levels*. They assume a Bell-LaPadula-like trust-model (compare Section 2.2.3). For external entities, a trust-level means the degree of trust that the external entity has. An assigned trust-level denotes the minimum trust-level for entry points and assets that an external entity has to have to interact with the entry point.

After the threat model has been collected, it is now examined for potential threats. The authors use the STRIDE approach for this purpose. The acronym

STRIDE stands for spoofing, tampering, repudiation, denial of service, and escalation of privileges. The categories correspond to a negation of the protection goals presented in the chapter on information security (compare section 2.1). For each model element, attacks in the categories mentioned above are now searched for. For example, the question is posed: *How can an attacker compromise the confidentiality of the data during communication between the external entity and the entry point?* Every found threat is sought out, and a risk assessment is carried out for it. The risk assessment involves a rough estimate of the damage, the effort required to carry it out and the effort needed to exploit the threat.

Swidersky and Snyder emphasise that threat modelling is not limited to any particular architecture modelling approach. However, they propose data flow diagrams because they claim that these diagrams have the advantage that they are not as strongly formalised as other modelling languages such as UML. Data flow diagrams and UML diagrams are explained in more detail in the section on modelling languages (see section 3.3).

Other sources took up the proposed approach (for instance, [143, 154]) after publication but criticised or improved various aspects. A frequently criticised aspect is the lack of tool support for threat modelling [14, 98, 130]. Shostack specified this aspect in his book from 2014 [143]:

> *There are two categories of features that people often ask for that are worth a brief discussion: automated model creation and automated threat identification. A great many people want tools that can take a piece of software that's already been written and extract a data flow or other architectural diagram. This is an attractive goal, and one that might be feasible for programs written in strongly typed languages.*
>
> [...] *Similarly, tools that can take a diagram or other model and produce lists of threats would be lovely.* [...] *However, these tools carry a risk that security analysis will focus only on known threats from an attack library. Such tools cannot (currently) analogize from closely related threats the way an experienced person can.*

2.3.3. Summary

Microsoft's approach to creating secure software is manifold. They enhanced all stages of their development lifecycle to be security-aware. Their effort has been rewarded with less discovered severe security vulnerabilities. Essentially, they turned security from an afterthought to a first-class citizen. One key factor to success is the threat modelling process that detects security flaws before the

software's implementation and helps one find architectural security flaws for legacy systems. Different sources state that threat modelling is a beneficial but time-consuming and expert-dependent step, and automation of threat modelling is essential.

2.4. Chapter Summary

This chapter gave a rough overview of security. The section on information security showed the different security properties that have to be protected to ensure an application's security. In this area, topics such as encryption, authentication and authorisation are of importance. For a long time, security was an afterthought that happened after a software system had been implemented successfully. To solve this problem, Microsoft's Security Development Lifecycle tries to integrate security into the development process to counteract the weaknesses that arise from the subsequent protection. According to Microsoft, threat modelling is a helpful tool here. Threat modelling tries to identify security errors at the architectural level. Then, possible mitigations against these mostly fundamental security errors are planned as early as possible. As others have noted, automating the threat modelling approach would be an essential step in advancing this idea. It is necessary to simplify or automate the creation of security architectures and then automatically examine them for security problems.

Chapter 3

Software Architecture

Software architectures are part of most classical software development processes. The idea of software architectures is to create an abstract model of the software to be implemented. Then, the model is a discussion basis for different software aspects, such as its structure, deployment, or security, with various stakeholders. Unfortunately, agile development processes neglect the importance of explicitly documented and planned software architectures due to their agile nature. In these approaches, the software changes very frequently, and written-down software architecture would result in many subsequent changes, slowing down development. Nevertheless, software architectures are still part of most security-related development processes, allowing security reviews of the software system.

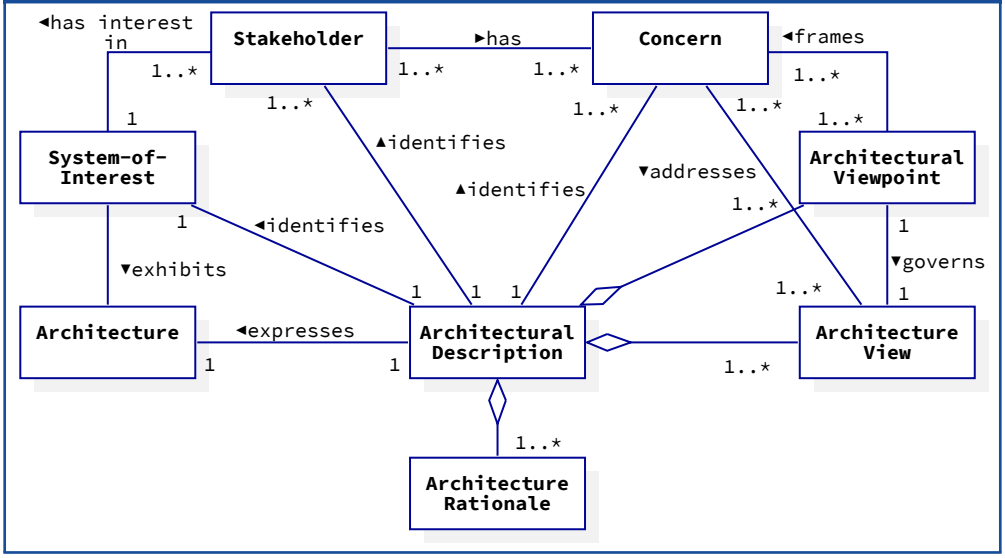
This chapter describes the necessary foundations of software architectures. First, Section 3.1 gives a brief definition of software architectures. Then, Section 3.2 explains different frameworks for architecture description that allow one to create and document software architectures systematically. Next, Section 3.3 shows an overview of so-called *architecture description languages* and approaches to representing programs. Finally, Section 3.4 introduces security patterns. Security patterns give reusable solutions to common security problems, and design patterns inspired them. Since the thesis deals with software security, this chapter primarily focuses on how the presented techniques deal with security.

3.1. Definition

Many different kinds of definitions of the term *software architecture* exist in literature (compare [28, 79, 82, 86, 108, 135], and others). A widely accepted definition is the one made by the Institute of Electrical and Electronics Engineers. IEEE standard 42010:2011 states that architecture consists of "*fundamental concepts or properties of a system in its environment embodied in its elements, relationships, and in the principles of its design and evolution*" [86].

Figure 3.1 depicts the base concepts of a software architecture as shown in IEEE 42010:2011.

Figure 3.1. Software architecture according to IEEE 42010:2011 [86]



According to this standard, an *architecture description* that contains different *architecture viewpoints* expresses an architecture. A set of viewpoints frames several *concerns*. The concerns originate from *stakeholders* who are interested in the *system of interest*. Each viewpoint governs an *architecture view* that consequently addresses different stakeholder concerns. Examples of concerns are features, reliability, cost, and modifiability. The standard mentions that security is a concern as well and, therefore, can be expressed as a separate view or incorporated into different views. Besides the *architecture description*, there is the *architecture rationale*, which explains design decisions' reasons to facilitate traceability. The IEEE 42010:2011 does not give a predefined set of *concerns* or *architecture views*, or a formal or graphical language to define a *software architecture*.

3.2. Architecture Description Frameworks

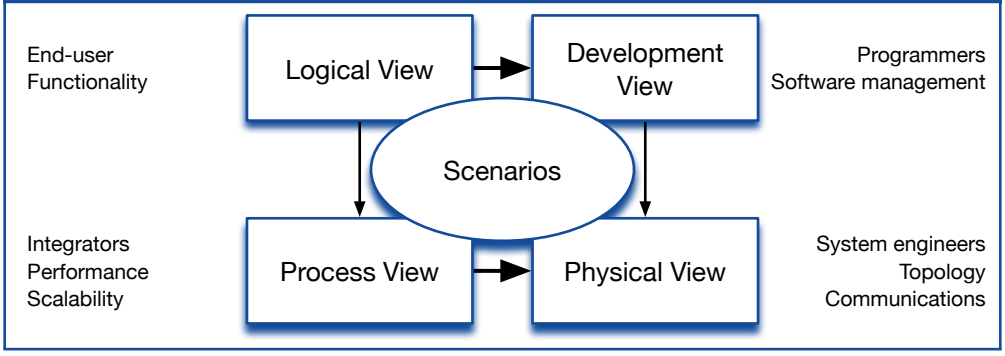
While IEEE 42010:2011 describes software architecture's essence, it does not specify which views to use or how to derive a software architecture for a system. Architecture description frameworks take on this task, and there are a number of these in literature. Each of these frameworks suggests different views that should,

in the autors opinion, be part of an architectural description. In most cases, these views are compatible with the architectural view of the IEEE 42010:2011 standard. In the following, this section presents different architecture description frameworks, and it examines whether and how they take the security of the software to be developed into account.

3.2.1. 4+1 View Model of Architecture

Philippe Kruchten published his *4+1 View Model of Architecture* in 1995 [102]. It consists of five distinguished views. He argues for a logical, a development, a process, a physical, and a scenario view. Kruchten stressed that it is unnecessary to use all views for every software system if there are reasons for omitting one or more of the views. For each view, Kruchten gives a graphical notation but notices that his views can be expressed by other notations as well. Figure 3.2 shows the relationship between the suggested views, and they will be described in the following.

Figure 3.2. The *4+1 View Model* of architecture according to Kruchten [102]



Logical View The logical view maps a software's functional requirements to the architectural components. Therefore, the system is decomposed into classes and connections between them. A class corresponds to a functional component, and the connections represent concepts, such as association, containment, usage, inheritance and instantiation.

Process View While the logical view contains the functional requirements, the process view takes the non-functional requirements into account. It addresses, e.g., the topics performance, availability, concurrency, and fault-tolerance.

In addition, the process view shows planned processes and the contained tasks or threads.

Development View The development view decomposes the system into subsystems and modules. A module is a small part of a system, one developer or a small team of developers can implement. The subsystems are ordered hierarchically and have a well-defined interface to the depending subsystems. Furthermore, the view contains references that stand for compilation dependencies.

Physical View The physical view maps the processes of the software system, defined in the process view, to the hardware. It shows the used processing nodes, the process assignments and the communication between the nodes.

Scenario View Scenarios describe the use cases of the system. For Kruchten, it is an additional view since it is redundant given the other views. This view uses similar concepts as the logical view.

The depiction shows how views influence each other. Kruchten does not define any cyclic dependencies. Therefore, it not possible that two views can influence each other, for instance, it is not possible that the *Physical View* influences the *Development View*. Use-cases, here called *Scenarios*, are the central driver of this approach from which all architectural elements are derived.

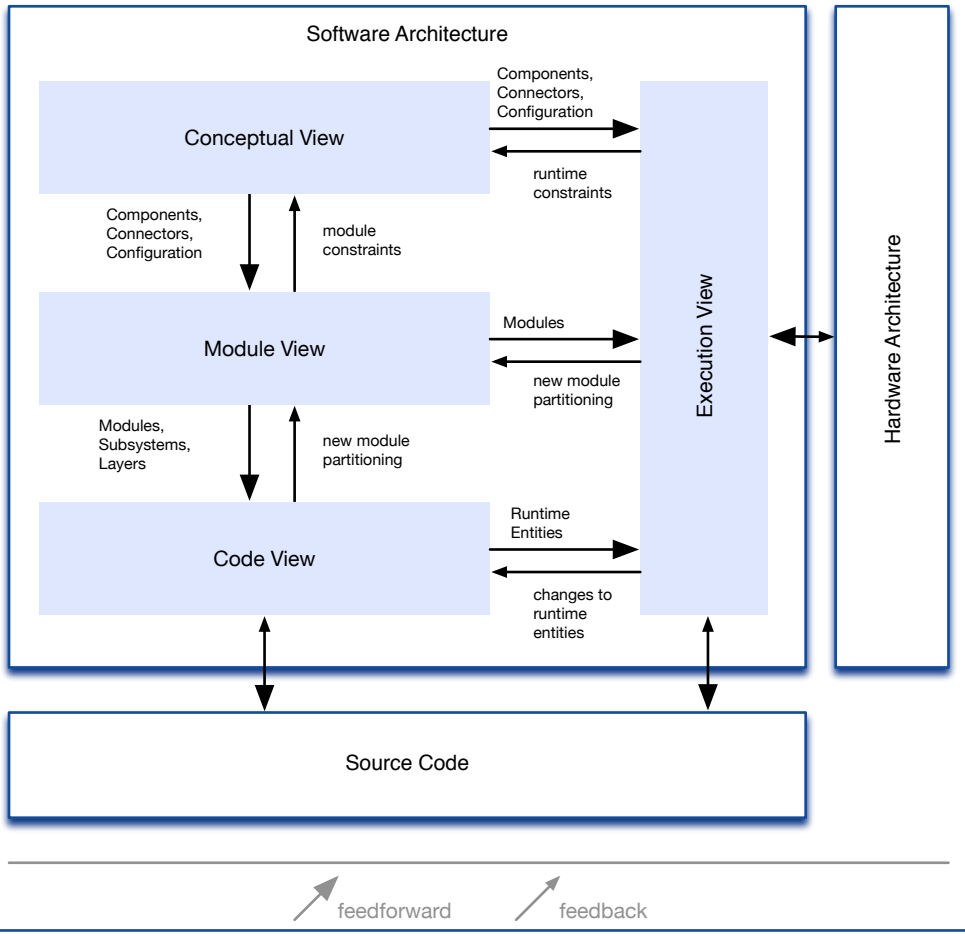
3.2.2. Hofmeister, Nord, and Soni

Hofmeister, Nord and Soni studied architectures of eleven mostly large industrial systems and proposed, based on their results, an architecture description framework consisting of four different views in *Applied Software Architecture* [82]. The remainder of this section describes the framework according to their book. Figure 3.3 depicts the suggested architectural views and their relationships.

The basic idea of the different views is to separate and decouple software engineering concerns. Each view tackles a significant software engineering matter and focuses on showing how the architecture solves it for the given software system. Before the views are created, Hofmeister et al. suggest conducting a global analysis. First, the global analysis identifies factors influencing the design of a software. Then, based on these factors, problems are identified, and strategies are developed on how an architecture can tackle the problems. Afterwards, the views described below are created.

Conceptual View The conceptual view is closest to the application's domain. It maps the system's functionality to conceptual components containing ports

Figure 3.3. Four architectural views according to [82]



that are coupled by connectors. A connector can show data or control flow between the modelled software components. The conceptual view shows, for instance, the integration of off-the-shelf components or how the system meets its requirements. The conceptual view's design goal is to minimise the impact of requirement changes on the software system.

Module View The module view maps the elements from the conceptual view to subsystems and modules and shows the realisation in the context of the used software framework or technology. The module view allows defining layers, modules, and interfaces. It shows the use of dependencies between modules and interfaces, as well.

Execution View The execution view shows the software's components allocation to the runtime platform and hardware architecture. Therefore, the view shows the system's runtime entities and their hardware expectations. In addition, the execution view explains how non-functional requirements, such as performance, are met.

Code View The code view maps the software's architectural components to deployment components, executables, and source components. The view helps developers to implement the software's architecture.

Hofmeister et al.'s architecture description framework does not explicitly address security. Therefore, security has to be incorporated into the four described views. Since security affects all levels represented in the views, it would also be necessary to divide the security aspects across all views. However, this division makes it difficult to overview all security measures because you have to identify these in different views.

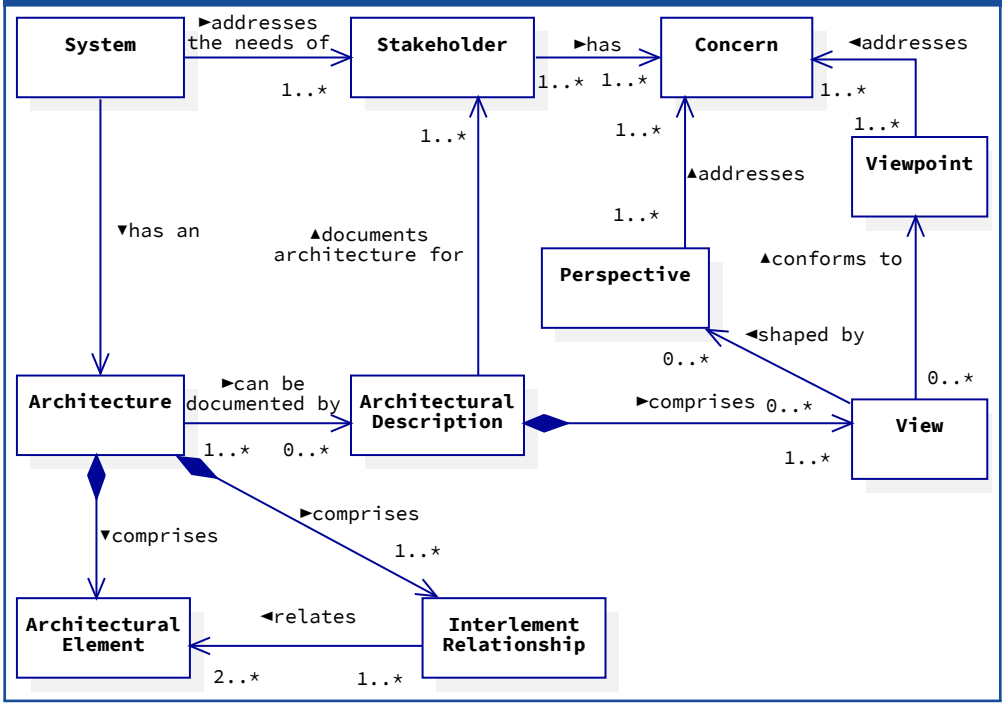
3.2.3. Rozanski and Woods

Rozanski and Woods use a slightly different definition of architecture and architectural descriptions. This section describes their view on software architecture according to their book from 2011 [135]. Figure 3.4 shows Rozanski and Woods conceptual model of software architectures. Besides minor naming changes and inverted associations, they added three new concepts. First, an architecture comprises a positive number of *Interelement Relationships* and *Architectural Elements*: "*An architectural element [...] is a fundamental piece from which a system can be considered to be constructed*". The interelement relationships relate two architectural elements to each other. Lastly, they added *perspectives*. Views are shaped by several perspectives which address different concerns. In the opinion of Rozanski and Woods, "*an architectural perspective is a collection of activities, tactics, and guidelines that are used to ensure that a system exhibits a particular set of related quality properties that require consideration across a number of the system's architectural views*". They introduced the perspective concept to explicitly describe the connections between different views.

Rozanski and Woods argue for six different views in their architecture description framework. These are the functional, information, concurrency development, deployment, and operational view and Figure 3.5 shows their relations.

Functional View Explains the functional architecture elements of a system. It lists the element's interfaces, usage, and responsibilities. The authors claim that the functional viewpoint is read first.

Figure 3.4. Architecture according to Rozanski and Woods [135]



Information View Shows the data stored, manipulated, managed, and distributed by the system. They argued that this viewpoint is critical because the purpose of any computer system is to process information.

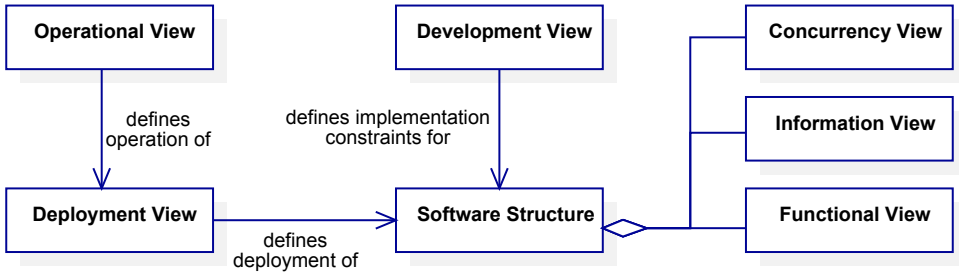
Concurrency View Describes the concurrency structure of the system and maps the different architecture elements to it. This viewpoint should answer the question of how the system is decomposed into several processes and threads.

Development View Explains the part of the architecture that supports the software development process. Therefore, it is essential for building, developing, testing, maintaining, and enhancing the system.

Deployment View Shows the deployment environment of the system under development. It maps the different processes and elements to runtime components where they will be executed.

Operational View Explains how the system will be operated, administered, and supported. This viewpoint should identify the operational concerns of the system's stakeholders and how they can be solved.

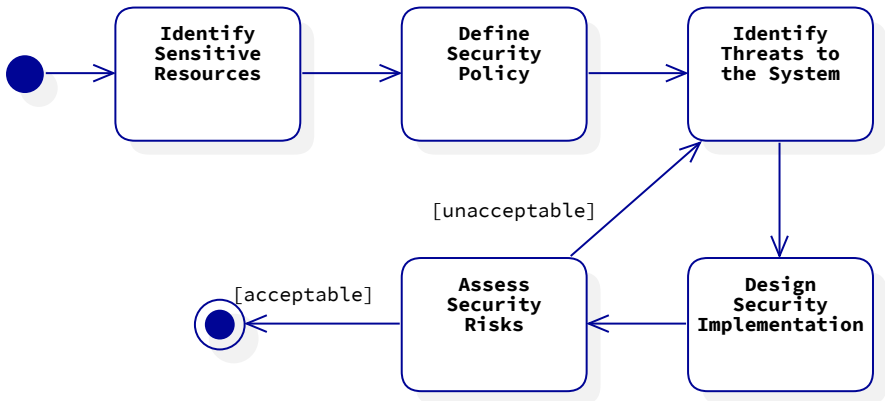
Figure 3.5. Architectural views according to Rozanski and Woods [135]



Rozanski and Woods introduced architectural perspectives to address cross-cutting quality concerns. These are necessary since quality requirements are orthogonal to the viewpoints mentioned above and impact multiple of them. Therefore, it is required to replicate the view's information, thus impeding its maintenance and synchronicity. To solve this problem, they proposed ten different perspectives, namely security, performance and scalability, availability and resilience, evolution, accessibility, development resources, internationalisation, location, regulations, and usability. They claimed that the security perspective, for instance, has a low impact on the concurrency viewpoint, a medium effect on the functional, information, development, and operational viewpoint, and high implications for the deployment viewpoint. Thus, a perspective defines addressed concerns and a set of activities necessary to understand the perspective's architectural impact Moreover, it represents a list of architectural tactics of known problems and pitfalls to the perspective, a checklist, and how to fulfil the quality requirements. According to the authors, perspectives are close to non-functional requirements. Nevertheless, they do not like the term since it suggests that these aspects are less important than the functional requirements.

According to Rozanski and Woods, the security perspective influences all six viewpoints. Figure 3.6 shows how to apply the security perspective. As a first step, they proposed to identify the sensitive resources of the system. The second step defines a security policy, documenting who is allowed to operate on what part of the sensitive resources. Afterwards, the system's threats are sifted out to discover the kinds of attacks the system needs to be protected from. Next, the security implementation is designed to mitigate the identified threats. The last step is evaluating whether the cost/risk balance of the security implementation is acceptable. If it is unacceptable, it is necessary to circle back to the threat-identification step. Otherwise, the perspective has been applied successfully.

Figure 3.6. Security perspective according to Rozanski and Woods [135]



3.2.4. Summary

This section gave an overview of different architecture description frameworks. While most approaches do not have separate views or concepts for describing security, Rozanski and Woods added a security perspective, emphasizing the importance of the security aspect. The other frameworks do not focus on security because they have been published at an earlier time when security did not play the role we see today. The topic of software security has become more and more relevant over the last two decades. Consequently, it has become more important to plan a secure software system and integrate security concerns more dominantly into the architecture. This goal is often referred to by the name *security by design*.

3.3. Description Languages

In literature, there are plenty of languages to describe various aspects of software systems. Besides language-specific graphical notations, the languages differ in the characteristics they can represent. Possible modelled aspects are the software's structure, behaviour, or interaction. Nowadays, there are general-purpose languages offering different diagrams for different characteristics, such as the Unified Modelling Language [122] and its extension Systems Modelling Language [122]. Furthermore, there are more specialised languages, such as the Business Process Modelling Notation for modelling business processes [119], Petri Nets for modelling and simulating the behaviour of software systems [124], or dataflow diagrams for describing the interaction of a system with its en-

vironment [56]. The remainder of this section describes different description languages used in more detail and focuses on their applicability to software security aspects.

3.3.1. Petri Nets

Carl Adam Petri claimed that he invented the concept of Petri nets at the age of 13 to describe chemical processes. He later formalised Petri nets in his PhD thesis [124]. This section is based on a publication by Desel and Juhás [61]. Today, the term Petri net is used for a specific class of graphs. In essence, Petri nets are directed graphs with typed nodes. One node type is *place*, while the second is *transition*. The arcs of a Petri net connect two nodes of different type. Additionally, there is the concept of markings in Petri nets. A marking is assigned to a node of type *place*. Some Petri nets have an initial marking, while others have initial transitions to produce markings. The Petri net can execute a transition if every place with an incoming connection to the transition has a required number of associated markings. The transition consumes the markings and produces markings at the targeted places.

Figure 3.7. Exemplary Petri net diagram

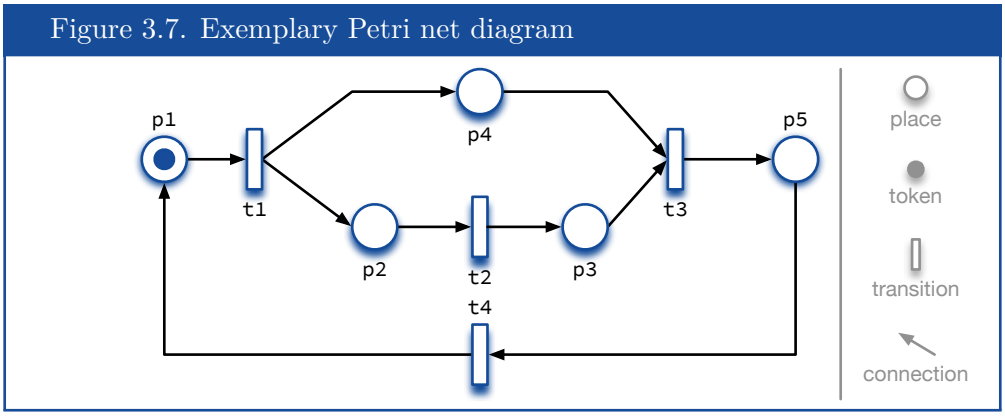


Figure 3.7 shows an exemplary Petri net with no practical meaning. It consists of five places, four transitions and an initial marker. First, the transition `t1` would be carried out by consuming the token from `p1`. For this, a token would be generated on both `p4` and `p2`. The transition `t3` cannot be executed at this point because `p3` does not contain a token yet. The Petri net can achieve the execution after `t2` is performed, and the token is transferred from `p2` to `p3`. Now, `t3` consumes a token from `p4` and `p3` to generate a new token on p5. After the transition `t4` has been executed, the Petri net is again in the starting state.

Petri nets are used to model processes and can be, due to their execution

semantics, executed. The execution can be used to identify if situations exist where the process halts unexpectedly because of modelling flaws. Furthermore, Petri nets have been used to check various security properties [74, 123, 172, 174].

3.3.2. Unified Modelling Language

The *Object Management Group* introduced the *Unified Modelling Language* "*to provide system architects, software engineers, and software developers with tools for analysis, design, and implementation of software-based systems as well as for mode[l]ling business and similar processes*" [121]. The remainder of this section gives an account of the UML specification. It consists of a formal definition of the abstract syntax and modelling concepts, a detailed explanation of the concept's semantics, and a human-readable graphical notation. The specification defines three different semantic areas: structural modelling, behavioural modelling, and supplemental modelling.

Structural modelling consists of means to describe the structure of a software system. It allows defining data types, classes, interfaces, signals, components, namespaces, relationships, and dependencies. At this level, the UML standard defines the following diagrams:

Package Diagram Package diagrams allow a modeller to divide a software system into several packages and sub-packages. The packages can have relations with each other, such as import relations. Package diagrams are suitable to describe architectural patterns, such as the layer architecture, the pipes and filters architecture, or the blackboard architecture.

Component Diagram Component diagrams give a black-box description of exchangeable architectural components. A component diagram represents the interfaces a component provides and consumes. The diagram does not express the behaviour of the described components. Therefore, it is necessary to use one of the behavioural diagrams described later. Nevertheless, it shows the dependencies of a component that is useful for implementation and planning tests.

Class Diagram The purpose of class diagrams is to describe classes, interfaces, and their relations and help detail the software system's underlying data model. Essential aspects of data models are the relations between the elements, multiplicity, and navigability. In object-oriented design, class diagrams are used to model the entities found in the real world to map them onto a software system.

Object Diagram While class diagrams allow to describe general concepts and their relations, it is often necessary to also capture concrete situations for instances of those classes. Object diagrams allow describing concrete instances of classes and their actual relations. Therefore, they help to explain the software system more precisely.

Describing a software system's actual behaviour is possible using the diagrams from the behavioural modelling level, which will be explained next.

Activity Diagram An activity diagram describes the workflow of a system or a component. It shows actions, as well as control flow and data flow between them. Störrle and Hausmann compared activity diagrams to Petri nets and described their close relationship [150].

State Machine Diagram A behavioural state machine describes different states of a component or software systems, existing transitions between the states and the events that allow switching between them. Furthermore, protocol state machines show the appropriate usage of interfaces or components. Both kinds of state machines can be expressed using state machine diagrams.

Sequence Diagram Sequence diagrams describe interactions between different objects based on timely-ordered occurring events. It is possible to express high-level interactions at the level of software components to explicitly describe concrete interactions at the class level using method interactions.

Communication Diagram Another possibility to show the exchange of events within a software system are communication diagrams. While sequence diagrams focus on the timely order of events, communication diagrams focus on the objects' collaborations.

Timing Diagram A timing diagram depicts the timely change between states of an object. It shows the timing behaviour of real-time systems.

The supplemental modelling level contains diagram types that neither fit into the structural nor the behavioural modelling:

Use Case Diagram Use case diagrams show the software's actors, the software's use cases, and their relations. Software architects can use use-case diagrams to describe the expected behaviour of a software system under planning.

Deployment Diagram It is possible to describe the deployment of software artefacts, e.g. components, using UML deployment diagrams. These diagrams are useful to show the allocation of a software system's components to different computers and the communication between these computers.

Information Flow Diagram A UML information flow diagram shows the flow of information between arbitrary elements or actors of a software system.

There are approaches in academia, such as SecureUML [112] and UMLSec [94], that extend the UML to include formal security aspects. These approaches make it possible to model these aspects, validate them in parts, or generate code from them. The related research chapter (see Chapter 13) gives a more detailed overview of these approaches.

3.3.3. Systems Modelling Language

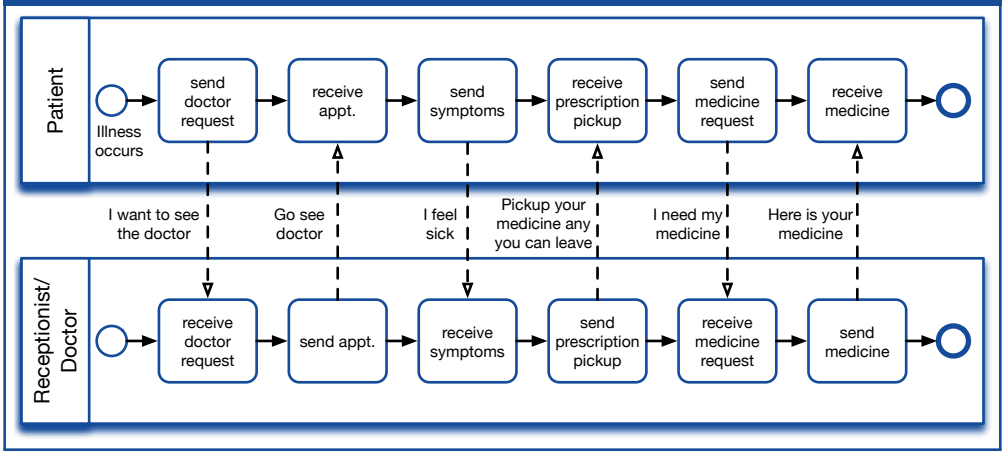
The Systems Modelling Language, also known as SysML, is a UML 2.0 profile for system engineers to meet the special requirements in the design of embedded systems [122]. It started in 2003 as an open-source specification, and the Object Management Group adopted it three years later. It reuses seven of the existing UML diagrams and adds new modelling aspects and diagrams to the UML 2.0. The reused but slightly modified diagrams are the use-case diagram, activity diagram, sequence diagram, state diagram, class diagram, and component diagram. In addition, the OMG added the parametric diagram and the requirement diagram. The parametric diagram allows defining constraints between system elements, such as the input or output range of electrical components. Besides, it is possible to model physical relationships to specify units of measurement and their conversions. Parametric diagrams make it possible to automatically check whether dependencies in the model adhere to these constraints. Although in the UML, there are use-case diagrams to describe the behaviour from the viewpoint of the system's stakeholders, SysML introduced requirement diagrams since many embedded systems, such as electronic control units employed to control cars, do not have users that directly interact with the system. Therefore, it is possible to document the system's requirements in the form of requirement diagrams. SysML allows refining requirements hierarchically and linking them to other model elements enabling traceability from the requirements to the design decisions. The traceability matches the concept of the architecture rationale described in the IEEE standard 42010:2011. Lastly, SysML adds allocation tables to explicitly map elements of different diagrams to each other.

3.3.4. Business Process Model and Notation

The content of this section is based on Freund and Rücker [75] and OMG's BPMN 2.0 specification [119]. Business process management aims to capture, execute, document, measure, and monitor business processes. A business process is a defined sequence of tasks to produce some value for the company's customers.

An example of a business process is lending at a bank, where potential customers are first advised, then have their credit check carried out, their credit terms set, and finally, their credit agreement concluded. It is possible that different subjects can carry out parts of these processes to comply with specific regulations. IT processes are the basis of more and more business processes and thus digitise companies. With business process management, the existing business processes of a company are collected and documented. Specified business processes help new employees to execute these processes. Furthermore, some certifications, such as the ISO 9000 [3], require companies to have documented business processes.

Figure 3.8. Exemplary BPMN diagram from OMG's BPMN 2.0 specification [119]



With the Business Process Model and Notation, the Object Management Group created a possibility to note, store, and exchange business processes [119]. Figure 3.8 shows an exemplary process taken from the specification. The BPMN diagram shows the process of visiting a doctor in case of an illness. It has one lane for the patient who wants to see the doctor himself and one lane for the doctor itself. These independent processes exchange messages to work together. The patient calls the receptionist to make an appointment for his visit at the doctor's. During the appointment, he describes his symptoms and receives a medicine prescription. The receptionist hands out the medicine based on the medication. If this process is mapped to a software system, one can derive that system's first set of security requirements from the BPMN diagram. There are two different roles involved in this system. Each role executes tasks exclusively, resulting in the requirement to protect sensitive tasks, such as the medicine

prescription.

3.3.5. Dataflow Diagrams

Dataflow diagrams have a long history in the area of computer science and software engineering. DeMarco first described dataflow diagrams in *Structured Analysis and System Specification* in 1979 [56]. DeMarco described dataflow diagrams (DFD) as a network of interrelating processes. He stressed that a dataflow diagram focuses on the flow of data, not on the control flow. According to DeMarco, dataflow diagrams consist of data sources and sinks, processes, files or databases, and dataflows.

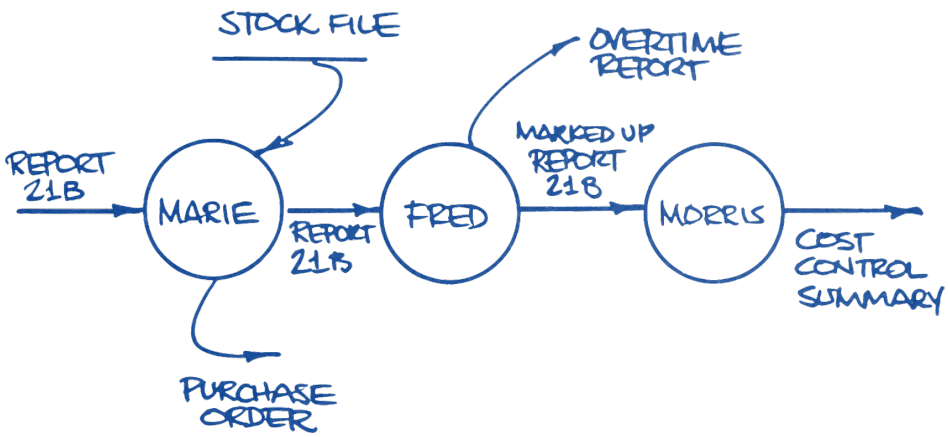
A named vector represents a dataflow in a dataflow diagram and is a pipeline for transferring information of a given form. The data flow specifies an interface between different parts of the diagram. In DeMarco's work, dataflows are allowed to either have no starting or no ending point. A bubble depicts a process that is a transformation of incoming data flows to outgoing data flows. Labelled lines denote files or databases. It is stated that a file is any temporary repository of data and not limited to the meaning of a file known in computer science. He sees databases as long-lasting data repositories. A box in a dataflow diagram stands for a data source or a data sink. Data sources and sinks are persons and organizations that are outside of the system's scope but interact with it. Additionally, DeMarco introduces procedural annotations to allow specification where incoming or outgoing dataflows are conjunctive or disjunctive. Furthermore, he described the idea of levelled dataflow diagrams to explain the internals of processes in greater detail.

Figure 3.9 shows a dataflow diagram from DeMarco's original publication. It shows the flow and processing of a report named *Report 21B* within a company. Initially, Marie receives the report and an additional temporary stock file. She then hands the report to Fred and creates a purchase order. Fred, in turn, processes the report, makes an overtime report and gives a marked report to Morris, who produces a cost control summary.

DeMarco's initial definition of dataflow diagrams was a lightweight and informal graphical representation of a system and the flow of data into, within, and out of this system. Over the years, different parties extended dataflow diagrams to express additional aspects or make them more formal. In 2008, Jilani, Nadeem, Kim, and Cho carried out a study to provide an overview of the modifications made to dataflow diagrams. They reported on 17 different research papers that modify the original concept of dataflow diagrams [88].

In 2004, Swiderski and Snyder described how Microsoft uses dataflow diagrams in their threat modelling process (please compare Threat Modelling in

Figure 3.9. Original Dataflow Diagram from DeMarco [56]

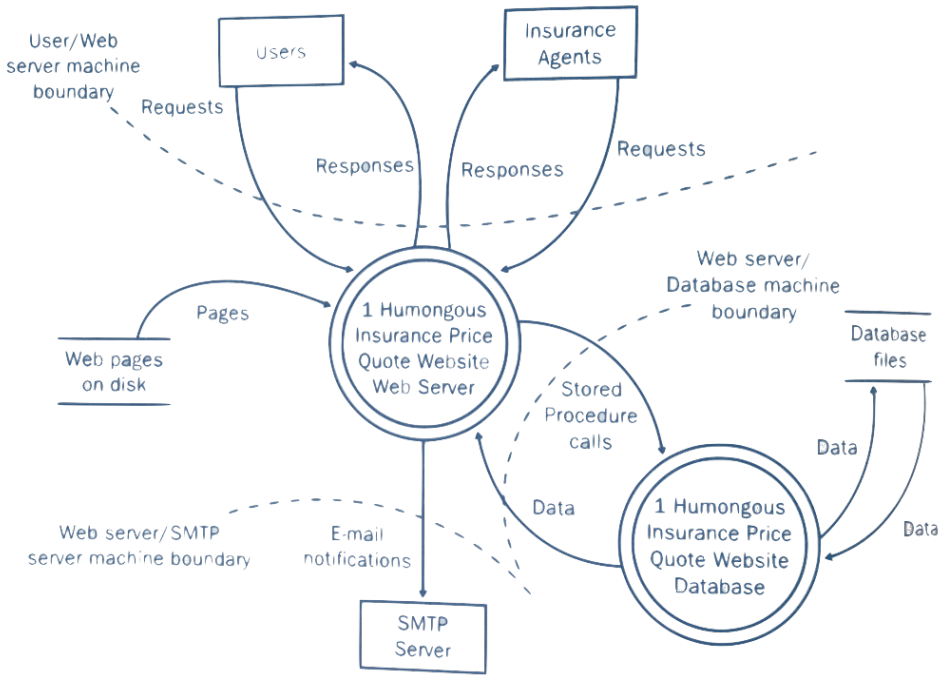


Section 2.3.2). This process is an attacker-centric approach for identifying threats and security flaws in a software system. Threat modelling employs dataflow diagrams for depicting *architecture views* addressing the security *concern*. For Swiderski and Snyder, one key feature of dataflow diagrams was the ease of using them and the lack of formal overhead. Nevertheless, they mentioned that threat modelling could be conducted with other notations, as well. The dataflow diagrams introduced in Microsoft's threat modelling were a modified version of DeMarco's dataflow diagrams as well.

The dataflow diagrams used by Swiderski's and Snyder's are very similar to those introduced by DeMarco. A circle with two outlines depicts a multiple process element. Data sources and sinks are called external entities. The file and database element is replaced by a data store that is represented by two lines instead of a single line. Lastly, they added privilege boundaries. A dashed line between two elements symbolizes that these elements have different privilege levels.

Figure 3.10 depicts an exemplary dataflow diagram given by Swiderski and Snyder. It shows a dynamic website system consisting of two processes. The *Humongous Insurance Price Quote Website Web Server* sends *stored procedure calls* to the *Humongous Insurance Price Quote Website Database*, which in turn sends *data* back to the webserver. The database sends *data* as well to the *Database files* and receives them from the files. The webserver sends *e-mail notifications* to an external *SMTP server* and receives pages from the disk. Furthermore, *external users* and *insurance agents* send *requests* to the

Figure 3.10. Dataflow Diagram from Swiderski et al. [153]

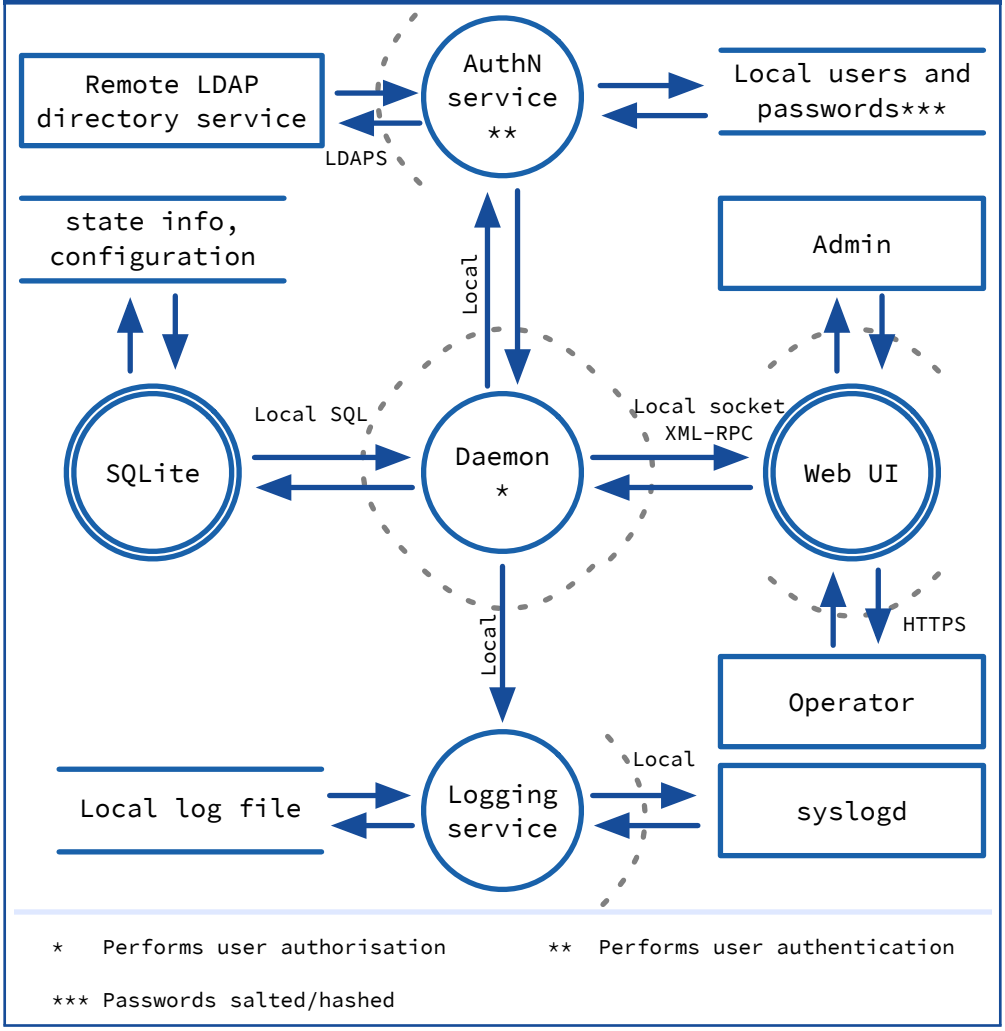


webserver and get *responses* in reply. Lastly, three different boundaries are shown, delimiting the machine running the webserver from other devices.

Dhillon, who was the principal security engineer at EMC Corporation's Product Security Office, described EMC's usage of threat modelling and dataflow diagrams in 2011. He noted that dataflow diagrams are easy and suitable to use but lack semantics. At the ECM Corporation, they started to annotate dataflow diagrams with information, e.g., privilege level, used programming languages, dataflow types, and encryption [63].

Figure 3.11 shows one of Dhillon's annotated dataflow diagrams. It depicts a demon process that is accessible to operators and admins via a web UI. The demon sends messages to a local log service, uses an authorisation service and an SQLite database. The database writes stale information and configuration information on a disk. The authorisation service talks to an external LDAP service and reads credentials from the disk. Finally, the logging service writes the log messages into a local log file and sends them to an external system logger, as well. In the depicted diagram, data flows are annotated with protocols instead of the sent data. Additionally, it is noted where the system authorizes and

Figure 3.11. Dataflow Diagram from Dhillon [63]



authenticates users and how stored data is secured.

As described, security experts use dataflow diagrams to represent security-related architecture views. Therefore, the security experts extended the original ones to have additional means to express security-relevant properties. Dataflow diagrams are, for instance, part of Microsoft's Threat Modelling approach, which Microsoft uses in their Secure Development Lifecycle. The purpose of these diagrams is to detect violations of all kinds of information security properties.

3.3.6. Summary

This section gave an overview of selected software description languages and how they can describe security properties. The languages range from general-purpose languages for software architectures with existing adaptations adding security information to very formal approaches used to prove security, and very informal approaches used in human-based processes to discuss software security from the architecture's point of view.

3.4. Security Patterns

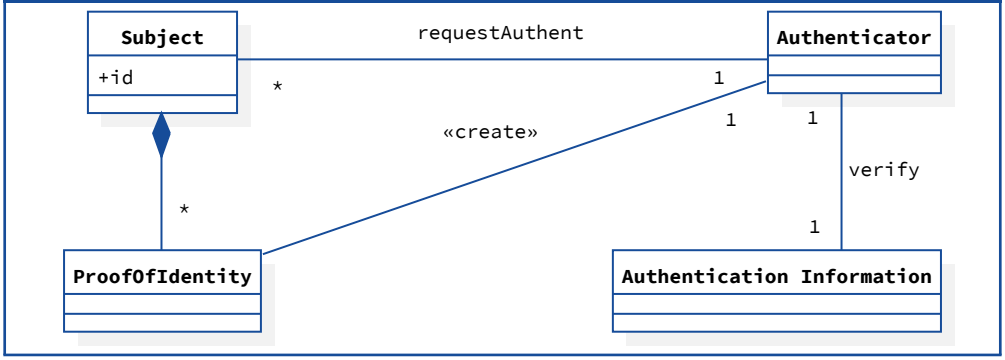
In software engineering, one possibility to deal with problems is the use of design patterns. The Gang of Four first published their book on design patterns in 1995 [77]. The basic idea is that developers and software architects can apply a design pattern every time a corresponding design problem arises during the creation of a software system. Furthermore, design patterns make code easier to understand because the behaviour of the pattern is clearly defined. The Gang of Four presented three pattern categories in their pattern catalogue: creational patterns, structural patterns, and behavioural patterns. Security research adapted the idea of patterns, and the topic of security patterns emerged [30, 45, 69, 70, 126, 171]. Some of these patterns are related to the protection goals introduced in the information security chapter and help to protect them which will be presented in the following.

3.4.1. Authentitactor Pattern

Eduardo Fernandez-Buglioni described the authenticator pattern in his book on security patterns [70]. According to Fernandez-Buglioni, the authenticator pattern verifies that a subject interacting with a system is the subject it claims to be. This definition aligns with the definition of authenticity given by the section on information security (c.f. Section 2.1).

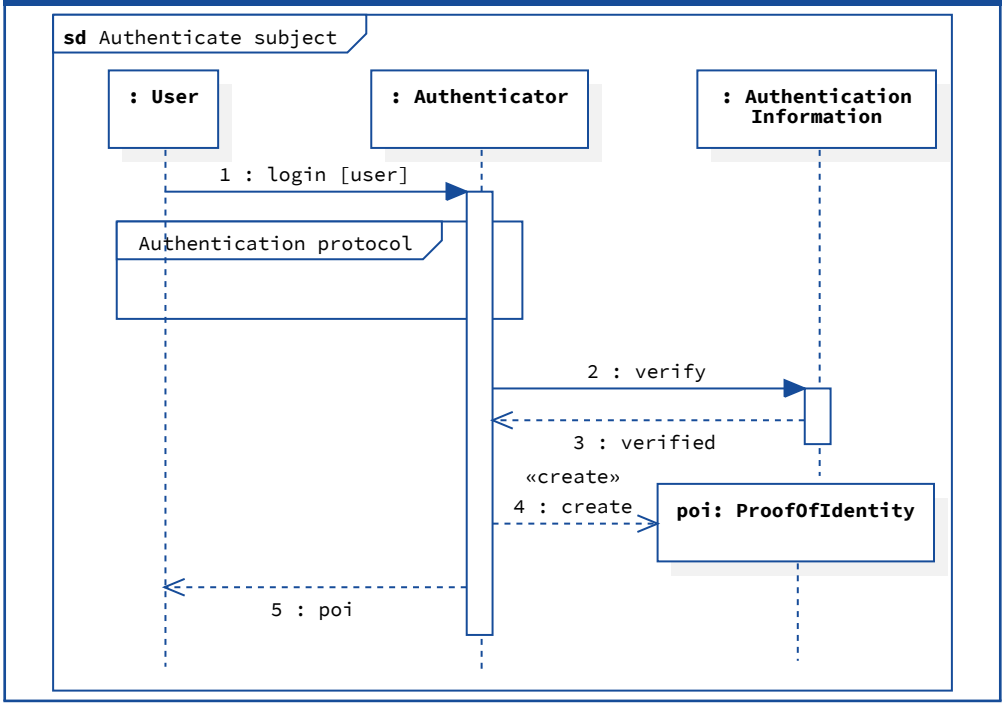
Figure 3.12 shows the authenticator pattern according to Fernandez-Buglioni. It consists of a `Subject`, the `Authenticator`, an `Authentication information`, and some `ProofOfIdentity`. Figure 3.13 shows the interaction of these elements. First, a user, which is a `Subject`, sends a `login` message to an `Authenticator`. Then, based on an authentication protocol, the user transmits authentication information to the `Authenticator`. The `Authenticator` then verifies the `authentication information`. If the verification is successful, the `Authenticator` assigns and returns a `ProofOfIdentity` to the subject.

Figure 3.12. UML class diagram of the authenticator pattern according to Eduardo Fernandez-Buglioni [70]



The `ProofOfIdentity` allows the system to check the identity during future requests.

Figure 3.13. UML sequence diagram of the authenticator pattern according to Eduardo Fernandez-Buglioni [70]



There are different realisations of the authentication pattern. In the case of

dynamic web pages, a common realisation is using an account identifier and a password, which are sent to the `Authenticator`. The `Authenticator` then compares the password to a stored password [1]. If the passwords match, a session identifier is generated and stored in the HTTP header and transmitted with each subsequent request. The web server can consequently check the session identifier on each request to determine the identity of the request's subject.

3.4.2. Authorisation Pattern[2]

Authorisation must ensure that a subject, e.g., a user or system, can access sensitive resources, e.g., some data or functionality. Enforcing authorisation policies consists of different aspects. XCAML [169], an XML-based access control specification language for XML, defines the following concepts:

PAP The *policy administration point* is a system part that updates the authorisation policy. It is not fixed to a specific access control model.

PDP A *policy decision point* is a component of the system that decides whether a subject can access some resource according to the authorisation policy. In the following, we call the checked property an *authorisation fact*. An *authorisation fact* is, for example, whether a user holds particular permissions or is assigned a specific role.

PEP The *policy enforcement point* is a system part that asks a PDP whether the current subject is permitted to access a specific resource and enforces the decision by granting or denying access. Please note that the *PEP* is not necessarily a single statement. The *PEP* consists of a *PDP* call and a conditional statement protecting the access of a sensitive resource. The *PDP* receives *authorisation facts* as parameters and returns an *authorisation decision*. At runtime, the *authorisation decision* holds the information if the current user holds the requested *authorisation facts*. The conditional statement then uses the *authorisation decision* to grant access to a sensitive resource. Therefore, the *authorisation decision* returned by the *PDP* has to flow from its call to the conditional statement. Another important flow is the flow of an *authorisation fact* to the *PDP* since it determines how the resource is protected.

PAP The *policy information point* gives access to the currently configured authorisation policy, which the *PAP* has created.

[1] Please note that it is not suggested to store the password directly since an attacker can try to access it. A better solution is to salt and hash the password to prevent an attacker from stealing it.

[2] The contents of this section have already been published at SCAM'2020 [34].

Figure 3.14. UML Component Diagram of the Authorisation Pattern

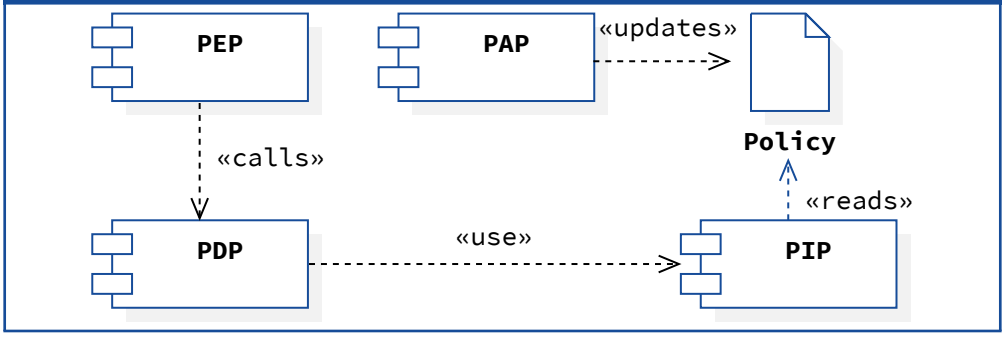
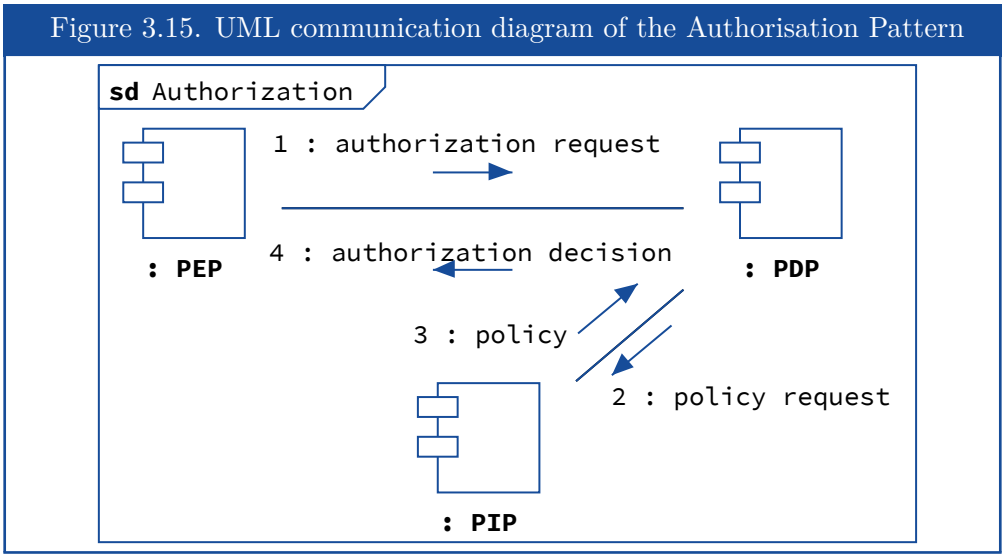


Figure 3.14 depicts the authorisation components and their usage dependencies in the form of a UML component diagram. In Figure 3.15, a UML communication diagram shows the dynamic communication of the elements mentioned above. First, a security administrator creates and stores an initial policy using the *policy administration point*. Then, at some later point in time, the *policy enforcement point* sends an authorisation request to the *policy decision point* containing the required *authorisation facts* to ask whether a subject can access a resource. In turn, the *PDP* asks the *policy information point* for the relevant policy parts and calculates the results for the request to answer this query. Finally, the *PDP* returns the authorisation decision to the calling *PEP*, which then grants or denies access.

Figure 3.15. UML communication diagram of the Authorisation Pattern



An implemented authorisation policy of a software system is the sum of all ac-

cesses to sensitive resources and the enforced *authorisation facts* on these paths. The intended authorisation policy, in contrast, is architectural information stating how each sensitive resource needs to be protected. During implementation, these two policies may diverge from each other. These differences can lead to potential vulnerabilities that allow unauthorised access to sensitive resources.

3.4.3. Cryptography Pattern

Braga et al. published the *Tropyc*, a pattern language for cryptographic software, in 1998 [44]. Using this pattern language, they presented nine patterns: *Information Secrecy*, *Sender Authentication*, *Message Integrity*, *Signature*, *Signature with Appendix*, *Secrecy with Integrity*, *Secrecy with Signature*, and *Secrecy with Signature with Appendix*. Since Braga et al. observed that these patterns have the same structure and behaviour, they extracted these aspects into a Generic Object-Oriented Cryptographic Architecture (GOOCA).

Figure 3.16. GOOCA structure according to Braga et al. [44]

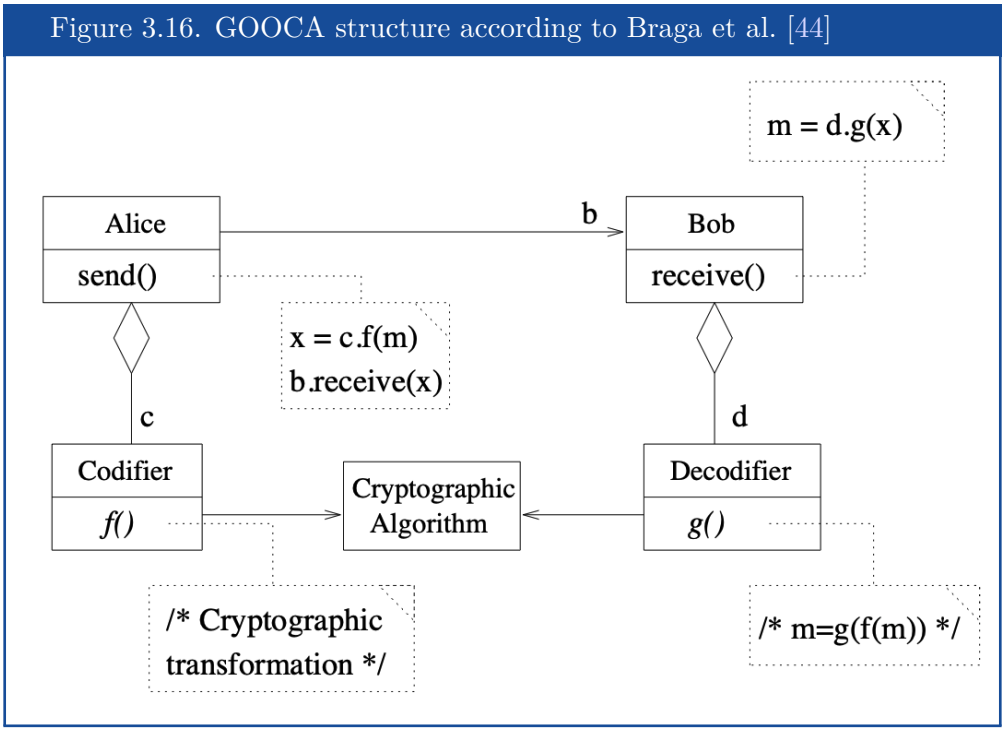
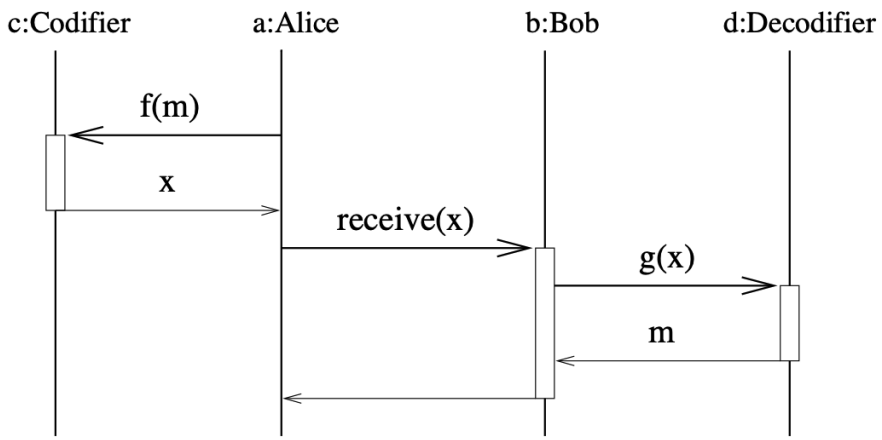


Figure 3.16 shows the structure of the Generic Object Cryptographic Architecture as described by Braga et al.. In essence, there is a sender called `Alice` who wants to communicate securely with `Bob`. Therefore, `Alice` uses a

`Codifier` and `Bob` a `Decodifier`. Whenever `Alice` sends a message to `Bob`, she first encodes the message using the `Codifier`, which in turn uses a `Cryptographic Algorithm`. `Bob`, in turn, decodes the message using his `Decodifier`. Figure 3.17 shows the dynamic behaviour of GOOCA.

Figure 3.17. Dynamic behaviour of GOOCA according to Braga et al. [44]



3.4.4. Summary

This section presented three security patterns related to the protection goals of information security. The patterns provide means to deal with authentication, authorisation, and cryptography. These aspects are necessary to protect authenticity, integrity, confidentiality, and non-repudiation. One fact that is obvious from these three descriptions is the difference in levels of pattern description details. Security patterns are an entire research field on their own and are investigated in other works, e.g. in the doctoral thesis by Bunke [45].

3.5. Chapter Summary

This chapter introduced software architectures according to IEEE 42010:2011 [86]. It gave an overview of different existing architecture-description frameworks. The purpose of an architecture-description framework is to have a standardised process of creating software architectures and a defined set of views that should be created. Only the framework proposed by Rozanski and Woods underlined the importance of software security by having an explicit security perspective.

Lastly, this chapter gave a brief overview of existing software description languages. It showed that in general-purpose languages, such as UML and SysML, there are no explicit means for expressing software-security aspects. Usually, UML stereotypes are used for adding security information. Instead, security aspects are modelled by either using mathematically founded languages, such as Petri nets or using relatively informal languages, such as dataflow diagrams. Finally, the chapter introduced security patterns, an analogous form of describing software patterns for security mechanisms. Section 3.4 concentrated on the security patterns authentication, authorisation, and cryptography as main security patterns since they tackle security topics introduced in the section on information security, c.f. Section 2.1.

Chapter 4

Static Analysis

Static software analysis attempts to reason about software without executing it. It ranges from finding text sequences in the program's source code to proving a software system's correctness. The roots of static software analyses lay in compiler construction, and both topics are still tightly coupled. Compiler construction developed techniques to read program code, transform it into more abstract representations, analyse the program, optimise the code, and generate machine-dependent executable code. Static software analysis analyses code to identify bugs and vulnerabilities or extract even more abstract representations from the software's source code or any related static information, such as configuration files. The term static analysis relates to the topic in general and to concrete static analyses as well.

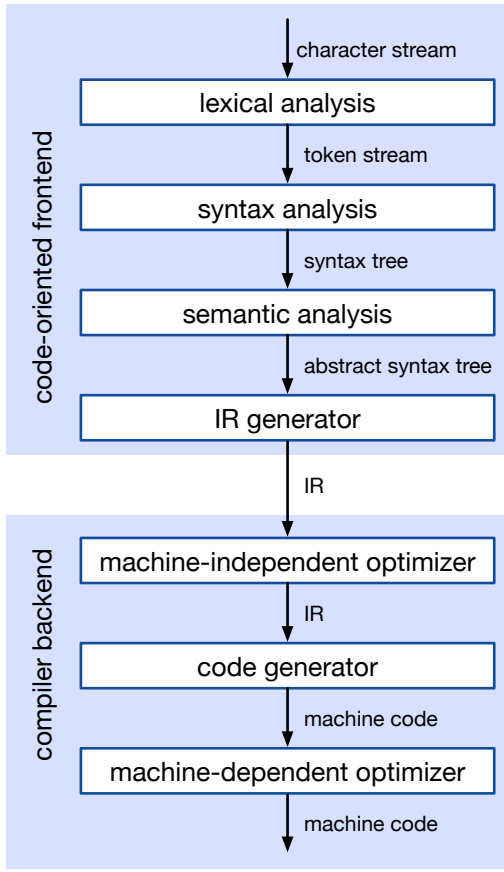
The structure of most higher-level static analyses is very similar to the one of a compiler. In fact, they often use the same techniques and representations. As already mentioned, static analyses differ in the level of abstraction they analyse on. Many of these levels stem from compiler construction. In some way, a compiler is a specialised static analysis that aims to generate executable code. In contrast, static analysis seeks to identify implementation-level bugs, detect high-level structures, or security flaws. Nowadays, compilers contain many static analyses to identify erroneous code, such as null dereference, buffer overflows or uninitialised variables detection.

This chapter's remainder explains the basics of compiler construction in Section 4.1. The following Section 4.2 focuses on different static analysis techniques. Lastly, Section 4.4 gives some details on existing security pattern detection techniques.

4.1. Compilers

This chapter follows the contents of Aho, Lam, Sethi, and Ullman's standard textbook on compiler construction [10]. The book contains the basis of the extended Figure 4.1, which shows the base steps of compilers.

Figure 4.1. Structure of compilers, based on Aho et al.



In a compiler, the lexical analysis first reads the source code character-wise and generates a stream of tokens. Afterwards, the syntax analysis takes place and constructs a syntax tree. The following semantic analysis transforms and enriches the syntax tree to an abstract syntax tree. The intermediate generator creates an intermediate representation (or short IR) based on the abstract syntax tree, representing the program in a suitable form for the following optimisation steps. All the mentioned steps take place in a source code-oriented frontend. These steps also take place in a source code based static analysis that works with an intermediate representation. In the compiler backend, a machine-independent optimiser reorganises the IR to improve the resulting executable code. While the representation's abstraction has increased in the previous steps, it now decreases again because the code generator now creates machine code for the

IR. In a final step, the machine-dependent optimiser optimises the machine code before finally writing the executable. However, not all static analyses are working with a source code-based frontend. Depending on the problem, it might read machine code or any kind of byte code, an intermediate representation used for virtual machines. A type of IR very frequently used in compilers and static analyses is the three-address code.

The remainder of this chapter explains the steps that take place in a compiler's frontend. Subsection 4.1.1 describes the lexical analysis. The following Subsection 4.1.2 presents the syntax analysis. Afterwards, Subsection 4.1.3 details the semantic analysis. Finally, Section 4.1.4 introduces the aforementioned three-address code and its generation.

4.1.1. Lexical Analysis

The lexical analysis reads a source code file character by character and uses it to generate a token stream. A token is a sequence of characters, and in addition to the textual representation, a token has a specific token type. The token type gives a token a meaning allowing one to interpret the textual representation. Typical type categories are comment, identifier, keyword, literal, operator, and separator. Besides, a token usually stores its position within the source code file as well. Depending on the analysed language, the so-called tokeniser often eliminates superfluous elements when processing the input. Such eliminated elements might include whitespace, as these have no meaning in programming languages[1]. Another commonly removed token type are comments if they are not of interest for later work.

It has long been customary to generate tokenisers automatically. A configuration file lists the regular expressions for all supported tokens. A scanner generator then automatically generates program code that automatically converts a text file into a token stream. Thus, the generator creates finite automata from the regular expressions, combines the automata into a single one, and then converts the resulting automaton into source code. At runtime, the generated tokeniser enters every character from the input file into the combined automaton. Every time the automaton is in an end state when reading whitespace, the tokeniser detected a valid token.

Figure 4.1 shows the source code of a simple Java class. It serves as the basis for all other subsections. The method `fibonacci` calculates the *nth* Fibonacci number using a recursive algorithm. First, the code declares a variable for the result. Next, it checks whether the method should calculate the first Fibonacci

[1]This statement does not hold for the programming language Whitespace.

Listing 4.1. Java-based Fibonacci implementation

```
class Fibonacci {
  /* calculates the n-th fibonacci number */
  public int fibonacci(final int n) {
    int result = 0;

    if(n == 1) {
      result = 1;
    } else if(n == 2) {
      result = 1;
    } else {
      result = fibonacci(n-1) + fibonacci(n-2);
    }

    return result;
  }
}
```

number. In this case, the value one is assigned to the result variable. This assignment happens as well if the following test of whether the method should calculate the second Fibonacci number is true. Otherwise, the sum of the recursive calls `fibonacci(n-1)` and `fibonacci(n-2)` is assigned to the result variable. In the end, the method returns the calculated result. According to the Java programming-language specification, the given example's calculated token stream contains more than 60 tokens.

Listing 4.2 shows a textual representation of the example's token stream. Please note that the formatting represents the source code's original formatting to support the understanding. Angle brackets surround each token containing the token type and the token's value if it is relevant. The tokens' source code location is not part of the representation to focus on the stream's relevant aspects. The generated stream is more abstract than the original source-code representation and allows the following syntax analysis to focus on the important information.

4.1.2. Syntax Analysis

The syntactic analysis processes the token stream of the tokeniser and derives a syntax tree from it. Part of this transformation is checking whether the program corresponds to the syntactic properties of the programming language. The language's grammar steers the syntactic analysis. Commonly, programming

Listing 4.2. Exemplary token stream of Listing 4.1

$\langle class \rangle, \langle ID, Fibonacci \rangle, \langle \{ \rangle,$
$\langle comment, calculates\ the\ n-th\ fibonacci\ number \rangle,$
$\langle public \rangle, \langle int \rangle, \langle ID, fibonacci \rangle, \langle (\rangle, \langle final \rangle, \langle int \rangle, \langle ID, n \rangle, \langle) \rangle, \langle \{ \rangle,$
$\langle int \rangle, \langle ID, result \rangle, \langle = \rangle, \langle 0 \rangle, \langle ; \rangle,$

$\langle if \rangle, \langle (\rangle, \langle ID, n \rangle, \langle == \rangle, \langle LITERAL, 1 \rangle, \langle) \rangle, \langle \{ \rangle,$
$\langle ID, result \rangle, \langle = \rangle, \langle LITERAL, 1 \rangle, \langle ; \rangle,$
$\langle \} \rangle, \langle else \rangle, \langle if \rangle, \langle (\rangle, \langle ID, n \rangle, \langle == \rangle, \langle LITERAL, 2 \rangle, \langle) \rangle, \langle \{ \rangle,$
$\langle ID, result \rangle, \langle = \rangle, \langle LITERAL, 1 \rangle, \langle ; \rangle,$
$\langle \} \rangle, \langle else \rangle, \langle \{ \rangle,$
$\langle ID, result \rangle, \langle = \rangle, \langle ID, fibonacci \rangle, \langle (\rangle, \langle ID, n \rangle, \langle - \rangle, \langle LITERAL, 1 \rangle, \langle) \rangle, \langle + \rangle,$
$\hookrightarrow \quad \langle ID, fibonacci \rangle, \langle (\rangle, \langle ID, n \rangle, \langle - \rangle, \langle LITERAL, 2 \rangle, \langle) \rangle, \langle ; \rangle,$
$\langle \} \rangle,$

$\langle return \rangle, \langle ID, result \rangle, \langle ; \rangle,$
$\langle \} \rangle,$
$\langle \} \rangle$

language's grammars are described using the extended Backus-Naur form. It has a long history and has been standardised by the International Organization for Standardization in 1996. According to ISO/IEC 14977, an EBNF noted grammar is based on three elements. First, there are terminals that correspond to the tokens of the tokeniser. Second, non-terminals are artificial elements for grammar specification. One of the non-terminals is the starting non-terminal that is the root element of the grammar derivation process. Finally, there are the production rules. A production rule specifies how terminals and non-terminals can replace a non-terminal. The replacement may contain special markers for repeating or omitting parts of the replacement. Starting at the initial non-terminal, the syntactic analysis tries to find a valid sequence of rule applications to consume all terminals, i.e. tokens. The analysed file is a correct source if the syntactic analysis can find a proper sequence. While processing, the syntactic analysis creates a parse tree, also known as the concrete syntax tree, based on the applied grammar rules. The tree consists of inner nodes and leaf nodes. An inner node corresponds to a non-terminal, and leaf nodes correspond to terminals. The root of the parse tree is the initial non-terminal.

There are different notions for writing down EBNF-based grammars. At the same time, they all express the same class of languages, namely context-free languages. They use different ways to set terminals, non-terminals, rules and how to set parts that can be repeated or omitted. Listing 4.3 shows a cutout

Listing 4.3. Excerpt from Java's grammar

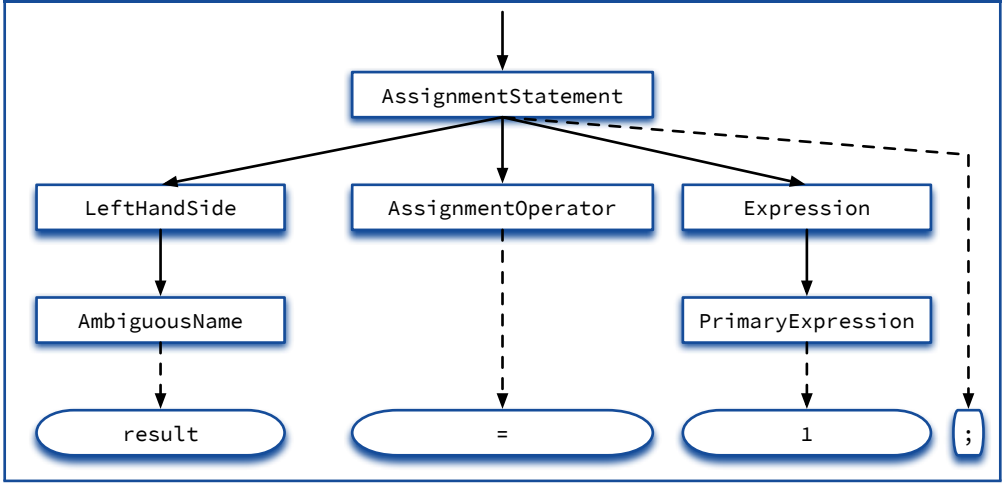
```
AssignmentStatement
  =  LeftHandSide AssignmentOperator Expression {AssignmentOperator
  ↪  Expression} ⟨;⟩;

LeftHandSide
  = AmbiguousName
  | ...;

AmbiguousName
  = ⟨ID⟩
  | ...;

AssignmentOperator
  = ⟨=⟩
  | ⟨* =⟩
  | ⟨/ =⟩
  | ... ;

Expression
  = PrimaryExpression
  | ...;

PrimaryExpression
  = ⟨LITERAL⟩
  | ...;
```

of Java's grammar[2] necessary to parse an assignment of the running example's method body. Left out derivations are represented by three dots. The listing uses the notation defined by ISO/IEC 14977. Every production rule starts with a non-terminal, followed by an equal sign. If there are several derivation rules for a non-terminal, these are separated from one another with a pipe. If there is a sequence of terminals and non-terminals in the replacement that can occur multiple times, it is enclosed in curly brackets. Each production rule ends with a semicolon. In contrast to the standard, the terminals are set in angle brackets to enable better readability with the previous subsection. The listing only focuses on the rules necessary to parse the assignment of a number to a variable. The rules start with the non-terminal `AssignmentStatement`.

Figure 4.2 shows the concrete parse tree for the instruction `result = 1;` from Listing 4.1 according to the grammar from Listing 4.3. The instruction

[2]The complete grammar is defined in the Java Language Specification [80].

Figure 4.2. Parse tree for statement `"result = 1;"` from Listing 4.1 according to the grammer given in Listing 4.3



corresponds to the token sequence $\langle ID, result\rangle, \langle =\rangle, \langle LITERAL, 1\rangle, \langle ;\rangle$ from Listing 4.2. The non-terminals of the grammar are contained in the tree as inner nodes in the form of rectangles. Terminals, the leaves of the tree, are shown as rounded rectangles.

As in the lexical analysis, a program can generate the syntactic analysis based on a specified grammar. Here, again, the problem is essentially reduced to an automaton that is automatically translated into source code. However, for the syntactic analysis, it is necessary to use a push-down automaton since, as already mentioned, context-free languages are involved. Generally, parser generators can generate code for both lexical and syntactic analysis since they are used together most of the time.

In this small example, the concrete parse tree already contains inner nodes that are only included due to the grammar and do not have any additional information, such as the `LeftHandSide` node. One part of the next step, the semantic analysis, is to remove these redundant nodes. Furthermore, it will check if the parsed source code is semantically correct. One of these semantic checks might be if the variable result has the correct type for an integer-literal assignment.

4.1.3. Semantic Analysis

While the previous analysis steps are very well automated, the semantic analysis checks properties where the automation is possible but less easy. The tests that

are possible in the semantic analysis very much depend on the implemented programming language. In this step, on the one hand, superfluous inner nodes are removed from the parse tree. On the other hand, the semantic analysis checks whether, for example, the typing of the expressions is correct according to the language definition. Depending on the abstract syntax tree's purpose, the nodes and types it consists of are different. More abstract information is sufficient for a compiler than for a refactoring tool since the generated code has to have structural similarity to the original code.

Checking the correct typing is only possible with typed programming languages. Furthermore, static type checking is not possible with many dynamic languages, such as JavaScript or Python. The type checking first assigns types to the leaf nodes with a definitive static type, for example, literals or variable types. Then, the type checking propagates the types upwards. During the propagation, the process checks whether the different children of an assignment have compatible types, according to the language definition.

Figure 4.3. Exemplary abstract syntax tree excerpt for the parse tree shown in Figure 4.2

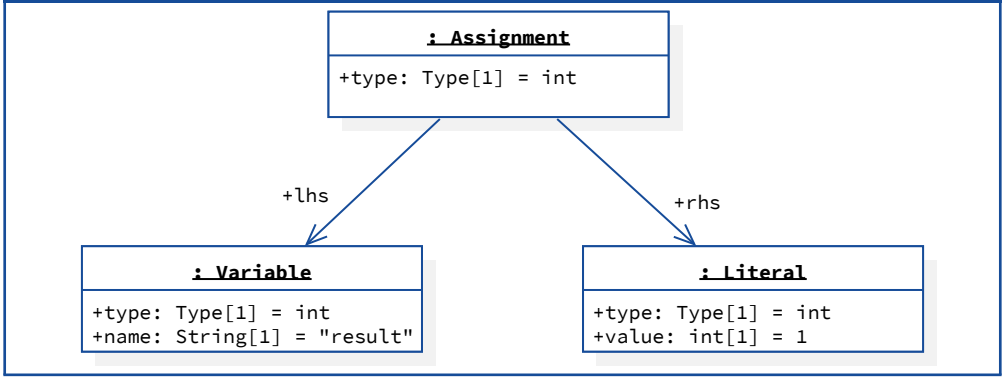


Figure 4.3 shows a section of an exemplary abstract syntax tree for the source text shown in Listing 4.1. Figure 4.3 focuses on the same assignment that Figure 4.2 shows. The abstraction reduced the number of nodes from ten to three, while some of the terminals have migrated to the nodes as attributes, such as the literal value. The ambiguous name has been resolved and converted into a node of type `Variable`. A type assignment was also carried out. Due to its declaration, the variable was given the type **`int`**. This also applies to the literal. This information was propagated to the assignment, where the semantic test successfully determined the compatibility from the left-hand side to the right-hand side of the assignment.

4.1.4. Intermediate Representation Generator

Programs consist of two different types of information—the type information and executable code. While the abstract syntax tree represents both information, the intermediate representation generator creates an intermediate representation for the executable code. The main reason for an intermediate representation is the possible complexity that grammars allow for programming languages. On the one hand, there are many different ways of expressing control structures, such as loops, and, on the other hand, expressions can be nested as deeply as desired. This complexity makes subsequent optimization steps difficult since a transformation has to take all eventualities into account. Therefore, the intermediate representation generator transforms the executable code into a more straightforward representation to simplify the following steps. An intermediate representation used very often is the so-called three-address code.

The name three-address code, also referred to by the abbreviation 3AC, results from the fact that each three-address code instruction uses a maximum of three operands. Complex source code instructions are therefore translated into several semantically equivalent 3AC instructions. While the three-address code has no explicit instructions for loops, they are represented with conditions and goto statements. The number of types of operations in the 3AC is also kept small to keep the subsequent steps' complexity as small as possible. As a consequence of this conversion step, the number of statements in contrast to the original program grows.

Listing 4.4 shows the Jimple code of the source code given in Listing 4.1. Jimple is the three-address code of Soot, a Java optimization framework, with a textual representation [162]. As introduced, the statements have up to three operands. The intermediate representation generator added labels used by the conditions as `goto` targets. Please note that the `goto` in an `if` statement is the `else` branch that is executed if the condition does not hold. In the case that the condition holds the statement after the `if` statement is executed.

The statement with the recursive calls "`result = fibonacci(n-1) + fibonacci(n-2);`" resulted in five different Jimple statements (from `label2` to `label3`). The transformation extracted the calculation of the parameter values, the calls and the addition of the resulting values into separate statements. It is noticeable that, in addition to the number of instructions, the number of necessary variables has also increased due to the transformation. A total of six variables are used in the three-address code, as opposed to three variables in the Java code.

Listing 4.4. Three-address code of the running example from Listing 4.1.

```
  r0 := @this: Fibonacci;
  i0 := @parameter0: int;

  if i0 != 1 goto label1;

  i5 = 1;
  goto label3;

label1:
  if i0 != 2 goto label2;

  i5 = 1;
  goto label3;

label2:
  $i1 = i0 - 1;
  $i2 = virtualinvoke r0.<Fibonacci: int fibonacci(int)>($i1);
  $i3 = i0 - 2;
  $i4 = virtualinvoke r0.<Fibonacci: int fibonacci(int)>($i3);
  i5 = $i2 + $i4;

label3:
  return i5;
```

4.1.5. Summary

This section introduced the basic principles of compiler construction, a research field strongly coupled to static analysis. Static analyses investigate programs to identify bugs and security flaws based on these steps and the data structures. The additional techniques and intermediate representations used in static analyses will be introduced in the next section.

4.2. Techniques used in Static Analyses

After introducing the fundamentals of compiler construction, this section presents data structures and algorithms that static analyses regularly use. First of all, Section 4.2.1 introduces control-flow graphs, followed by the explanation of callgraphs in Section 4.2.2. While these analyses deal with the control-flow aspects, the remaining sections deal with dataflow. Section 4.2.3 focuses on intra-procedural dataflow analysis, while Section 4.2.4 continues with inter-

procedural dataflow. Finally, Section 4.2.5 summarises this section.

4.2.1. Control-flow Graphs

The abstractions presented in the last chapter closely mirror the source code's structure and hence are very linear. However, control flow within the source code is not linear at all. Therefore, control-flow graphs focus on the representation of control within a method or function. In essence, a control-flow graph contains vertices for executable code and edges to represent the execution order of statements. Static analysis frameworks can generate such a graph directly for a method's abstract syntax tree or, such as Soot does, for the intermediate representation.

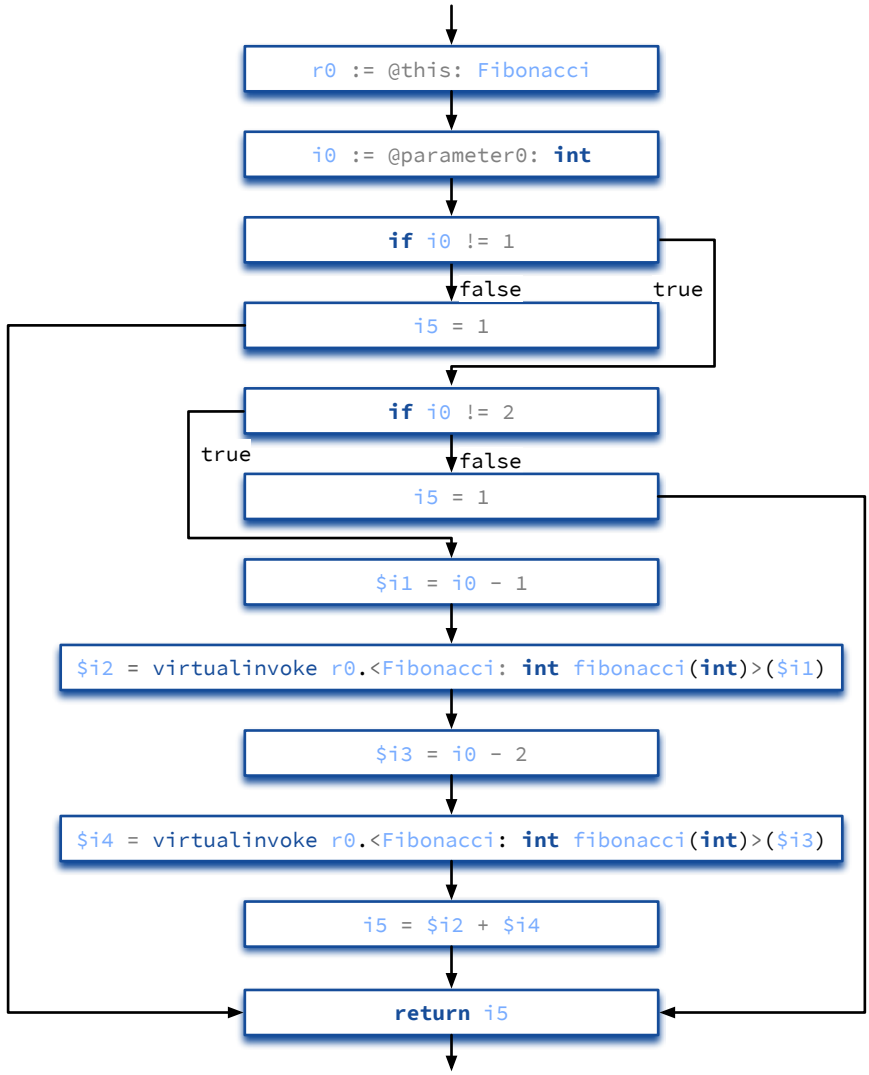
Figure 4.4 shows the control-flow graph of the Jimple code introduced in Listing 4.4. The topmost vertex is the entry of the method, and the control flow follows the edges. If a node has more than one successor, the edges are labelled with guards, representing the condition that must hold for the edge to be followed. The vertex at the figure's bottom is the exit node of the control-flow graph. While a control-flow graph can have multiple exit vertices, there can be only a single entry vertex. The shown control-flow graph ignores exceptional control-flow, but it is possible to extend control-flow graphs with information on exceptional control-flow.

4.2.2. Callgraph

A crucial basic data structure used in inter-procedural analyses are callgraphs. They represent the call relations between different functions and methods of a program. Thus, they have a significant impact on the analysis results that build on top of them. However, the calculation of callgraphs is, for different reasons, complicated and not even feasible. To calculate a callgraph, it is necessary to calculate points-to information, which reduces to the halting problem and therefore is not decidable in general and approximations are necessary. Such approximations allow to find $\mathcal{NP}$-complete solutions to static analysis problems [106, 127]. Since callgraphs are important to many existing problems, the literature describes different algorithms for calculating them. These algorithms differ in their time and memory requirements at the cost of preciseness.

While calculating callgraphs looks easy for static function invocations, it gets complicated in the presence of pointers and when object-oriented languages are used. The problems for calculating a precise callgraph are, on the one hand, facts that are not statically determinable, such as user input, and on the other hand, virtual method calls, where the called method depends on the

Figure 4.4. Exemplary control-flow graph for the intermediate representation of method `fibonacci` shown in Listing 4.4



type of the base object. As a result, a statically calculated callgraph always overapproximates. The result of a callgraph construction algorithm depends on how it builds contexts to abstract a system's state. The most imprecise callgraph is a context-insensitive callgraph where the calculation algorithm does not consider the system's state.

The class hierarchy analysis (CHA) [55] is the most common context-insensitive callgraph construction algorithm used for object-oriented systems. The basic idea is to resolve the targets of a method call by analysing the class hierarchy of all defined classes. Then, based on the class hierarchy and the programming language semantics, the analysis chooses possible targets of method calls. A refinement of the CHA analysis is the rapid type analysis (RTA) [22] proposed by Bacon and Sweeny. The main idea of this approach is to collect all the types instantiated in a program to exclude methods that are not reachable because the program does not create the corresponding types.

A context, in essence, is a model of the system's state. Context-sensitive algorithms map variables and class attributes to objects, constants, or any other form of abstraction. The term context-sensitive relates to the fact that such algorithms distinguish method calls and method executions based on the context. The algorithm assumes that a method behaves functional and the behaviour of a method is the same if it is called in the same context. The number of mapped facts and the used definition of context equality determines the memory and time consumption besides the program's size and complexity.

The variable type analysis (VTA) [152], for instance, is a simple context-sensitive callgraph construction algorithm. The algorithm collects a set of types of the objects a variable in the program can point to. Then, based on the calculated sets, the algorithm determines the targets of method calls. While the VTA outperforms CHA and RTA in terms of preciseness on the cost of resource usage, there are still possibilities to use more precise abstraction to generate even more precise callgraphs.

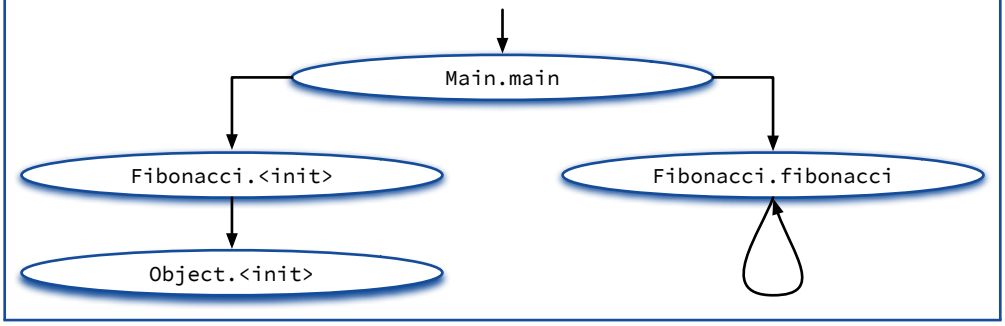
Listing 4.5. Exemplary main method using the class shown in Listing 4.1.

```
class Main {
    public static void main(final String **args) {
        final Fibonacci obj = new Fibonacci();

        final int value = obj.fibonacci(4);
    }
}
```

Listing 4.5 shows an exemplary main method using the fibonacci calculation shown in Listing 4.1 to calculate the fourth fibonacci number. CHA, RTA, and the presented VTA will all calculate the same callgraph of the application's code[3]. Figure 4.5 shows the calculated callgraph.

[3]Please note that the shown code does not call framework methods, such as from the JDK,

Figure 4.5. Exemplary context-insensitive callgraph for the code shown in Listing 4.5 and Listing 4.3



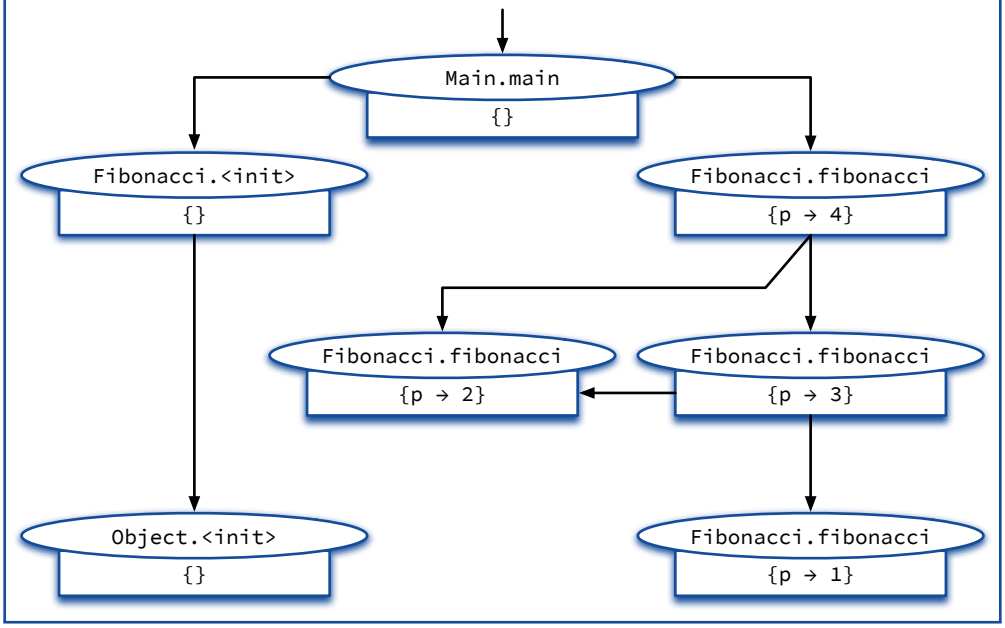
The named algorithms do not differ since no inheritance is present in the example. Suppose the used context tracks constant values, and additionally, the callgraph construction algorithm is flow-sensitive and path-sensitive, which means that the analysis respects the order of statements and the effects of conditions. In that case, it is possible to compute a more precise callgraph, as shown in Figure 4.6. If the algorithm is not path-sensitive, it cannot rule out which branches within the method `fibonacci` are feasible. This circumstance would lead to the situation that the analysis creates a very large amount of contexts since the value of the parameter `p` would be decreased as long as possible. Nevertheless, the algorithm would terminate eventually since there is only a finite number of **`int`** values, but the amount of required memory would be enormous. This small example already shows the problems involved in callgraph generation, even if only one critical aspect, the constant values, is considered.

4.2.3. Intra-procedural Dataflow Equation

In 1973, Kildall proposed a still used approach to calculating dataflow-based information at the intra-procedural level [99]. The proposed approach can compute information, such as constant propagation or intra-procedural reaching definitions. The approach is the base for different algorithms still used nowadays and works with a control-flow graph. For the algorithm, it is necessary to calculate a `gen`-set and a `kill`-set for each statement in the CFG. The `gen`-set contains the flow facts a vertex produces, while the `kill`-set contains the flow facts a vertex kills. Listing 4.6 shows the pseudo-code that explains the principle algorithm.

to remain manageable

Figure 4.6. Exemplary context-sensitive callgraph for the code shown in Listing 4.5 and Listing 4.3



Due to the precalculation of the `gen`-set and `kill`-set, it can be shown that the algorithm terminates in all cases. Since the `gen`-sets and `kill`-sets do not change during the analysis, the calculated `in`-sets for all vertices of the control-flow graph remain finite. An `in`-set contains the flow facts holding before entering a vertex and is created by combining all flow facts produced by preceding vertices of the control-flow graph. For combining these flow facts, it is necessary to specify a confluence operator. The algorithm terminates if the calculated flow facts do not change anymore and thus reaches a fixpoint. The idea presented by Kildall can be used to propagate information, such as constants or reaching definitions, but it has its limitations. For instance, the algorithm can propagate constant literals. Still, it is impossible to calculate new constant values based on the propagated literal values due to the algorithm's precalculation of gen-sets.

Over the years, the algorithm has slightly been modified to overcome this limitation. Instead of precalculating the `gen`-sets and `kill`-sets, the effects of a vertex are calculated during the propagation phase, allowing the algorithm to, for instance, add two constant literals found in the source code and propagate the calculated constant value for further use. While this approach is more powerful,

Listing 4.6. Pseudo code of Kildall's algorithm

```
// initialise out-sets
for (v ∈ cfg) {
    out[v] ← ∅;
}

// initialise working list
working ← {entry};

// process vertices while working list is not empty
while(working ≠ ∅) {
  current ← working.pop();

  // calculate new in-set based on predecessors
  newIn ← ∅
  for(v ∈ current.predecessors()) {
    newIn ← newIn ∪ out[v];
  }

  // skip vertex if in-set remains the same
  if(newIn = in[current]) {
      continue;
  }

  // store new in-set
  in[current] ← newIn;

  // calulate new out-set based on new in-set, kill-set, and gen-set
  out[current] ← (newIn \ kill[current]) ∪ gen[current];

  // add all predecessors to working set
  working ← working ∪ current.successors();
}
```

it is necessary to ensure that the propagated information remains finite and that the flow facts grow strictly monotonically. The finite nature poses a problem in the constant value propagation example since the int value can be incremented within a loop. Thus, the value can grow infinitely as long as the algorithm does not limit the contexts created for a node. A solution to ensure such problems do not occur is to use a bounded lattice as flow information.

4.2.4. Inter-procedural Dataflow Analysis

It is not sufficient to use intra-procedural analyses for many problems. That is the point where inter-procedural analyses come into play. There are different approaches to solve interprocedural analysis problems. One example is the reduction of inter-procedural finite, distributive subset dataflow (IDFS) problems to a graph reachability problem presented by Reps et al. in 1995 [131]. This approach allows an efficient and fast implementation, such as provided in Heros [41], but cannot solve all kinds of inter-procedural dataflow analyses since the problem must be a finite, distributive subset problem. This means that the flow facts must be finite and distributive over the used *meet*-operator. This approach allows, for instance, to calculate if a variable is constant at compile-time, while it is not possible to calculate the constant value.

An approach that is more general but not as efficient as the one by Reps et al. is the usage of value contexts as presented by Padhye and Khedker [141]. They use the functional approach to inter-procedural dataflow analyses proposed by Sharir and Pnueli [141]. This functional approach uses a callgraph to connect the control-flow graphs of different functions. The principle idea is that a method with a specific calling context behaves functionally and will produce the same result whenever it is called with that calling context. When finding a call expression, the algorithm calculates an input dataflow value—the calling context—for the call and computes an output dataflow value for the method by respecting the facts of the underlying control flow graph. The algorithm then stores the input and output value pair for the method and reuses the output value if it finds a call with the same input dataflow value.

It is possible to conduct inter-procedural dataflow analyses that are non-distributive over the *meet*-operator using the value context approach. An example of such a problem is the non-distributive sign-analysis that determines if a numeric variable is positive, negative, zero, or unknown.

4.2.5. Summary

This section introduced different techniques and basic concepts used in static analyses. These presented techniques help to take control flow into account and to calculate intra-procedural and inter-procedural dataflow. Furthermore, these fundamental techniques help realise complex, problem-specific analyses supporting to reason about different aspects of a software system.

4.3. Architecture Recovery

Architecture recovery tries to extract a given system's architecture automatically, and there is a plethora of research in this area [64, 78, 140, 173]. The remainder of this paper reflects Ducasse's and Pollet's summary of software architecture reconstruction [64]. The classification focuses on static architecture recovery techniques since it only mentions dynamic information as a possible input.

Ducasse and Pollet point out that there are two different kinds of architecture in literature: conceptual architecture and concrete architecture. The conceptual architecture is also known as *intended*, *as-designed*, or *logical* architecture. The conceptual architecture is planned in the design phase and is often written down using architecture description languages. The concrete architecture, also known as *implemented*, *as-built*, or *physical* architecture, is found in the program's source code. In many cases, there are differences between the conceptual and the concrete architecture of a software system. According to the authors, there are different reasons, techniques, and goals in reconstructing concrete architectures. They created a taxonomy to categorise existing research. Therefore, they chose the categories of *input*, *goals*, *processes*, *techniques*, and *output*.

The authors write that software architecture recovery always serves a purpose. They identified six different *goals* for why recovery is performed:

Redocumentation and Understanding Aims to redocument a software's concrete architecture and design decisions. Redocumentation and understanding are necessary for legacy systems that evolved without planning an architecture explicitly. Here the goal is to help with further development of the software by creating an architecture.

Reuse Investigation and Product Line Migration Focuses on identifying commonalities and variabilities between different products. This information allows migrating towards an underlying product line architecture.

Conformance Highlights differences between the conceptual and the concrete architecture by comparing them. Therefore, the concrete software system's architecture needs to be recovered.

Co-Evolution Tries to allow a parallel evolution of a software's conceptual and concrete architecture since they are two different abstractions of the same thing. Co-evolution synchronises the evolution by showing differences between them by adapting one or the other.

Analysis Wants to analyse a software's concrete architecture, for instance, regarding quality aspects, dependencies, or style conformance. The software

architecture recovery is simply the first step to be able to do so.

Evolution and Maintenance Investigates a concrete software architecture's possibilities to allow the software's evolution and maintenance. Some approaches even take a look at the software's concrete architecture history.

Related and Orthogonal Artifacts Extracts information related to the software's architecture, such as design patterns, features, aspects, roles, or collaborations. This information is not part of architecture, views or viewpoints, but it provides valuable information.

In the *processes* category, Ducasse and Pollet distinguished three different approaches. First, the bottom-up process starts at the source-code level and step-wise merges elements into more abstract architectural components. Second, the top-down process starts with high-level knowledge and matches them to source-code elements. The architecture artefacts used here are requirements or architectural styles. Lastly, hybrid processes use both approaches and work with architectural and source-code information. The architectural information is refined while the ource-code information is abstracted until they have the same abstraction level.

According to Ducasse and Pollet, there are two main categories of *input*. First, there are architectural inputs, such as architectural styles and viewpoints. Second, there are non-architectural inputs, such as source code, textual information, e.g., comments, dynamic information, physical organisation, human organisation, historical information, and human expertise. Top-down and hybrid approaches use architectural information. Additionally, they can use some non-architectural information, such as human expertise, historical information, or organisation information. Bottom-up approaches mainly work with the software's source code, textual information and dynamic information.

The authors divide the software architecture recovery *techniques* into three main categories: *quasi-manual*, *semi-automatic*, and *quasi-automatic*. Quasi-manual techniques help reverse engineers to identify architecture elements manually. In this category, the authors distinguish between construction-based and exploration-based techniques. The first category supports the manual creation of the software's architecture, while the second category guides the reverse engineer through the system's high-level artefacts. The semi-automatic techniques divide into two main categories: abstraction-based techniques and investigation-based techniques. Therefore, abstraction-based techniques map low-level concepts to high-level concepts with relational queries, logic queries, hard-coded programs, or lexical and structural queries. The investigation-based techniques

map high-level concepts to low-level concepts. They use recognisers, graph pattern matching, state engines, or maps. Finally, the quasi-automatic techniques group objects that are similar according to a specific definition of similarity. The quasi-automatic techniques utilise concept analysis, clustering algorithms, dominance information, and layers and matrix. The difference between the three mentioned main categories is the degree of control a reverse engineer has. In the first group she has to identify the components manually. In the *semi-automatic* group, the reverse engineer instructs the tool to automatically discover components, and in the last group the tool has the control and the reverse engineer instructs the process.

Finally, Ducasse and Pollet distinguish between four different kinds of outputs. First, *visual software views* are meant to represent the software's architecture in the form of box-like architecture description, source entity visualisation, or more sophisticated architecture views. Second, *architectures* in the form of architecture description languages to use them in architecture-centric development. Third, architecture *conformance* to identify differences between the software's conceptual and concrete architecture. Lastly, the authors mention the storage for additional *analysis*, such as the calculation of quality metrics.

4.4. Security Pattern Detection

Van Hilst and Fernandez introduced in a position paper in 2007 the idea of using reverse engineering, particularly pattern detection techniques used for the detection of design patterns, to identify security patterns in code. They compare security patterns to the design patterns introduced by Gamma et al. Therefore, they use the state and the strategy patterns from Gamma et al. [77] and compare them to authentication and access control security patterns. The authors point out that the size of design patterns and security patterns differ, making security patterns more complex to detect. They state: "*Security patterns differ from GoF patterns in size, and in the specific features most likely to be distinguishing characteristics in a match. However, many of the techniques presented seem well suited to the task*" [163].

In 2011, Bunke et al. presented their work on checking the usage of security patterns using the hierarchical reflexion analysis (the hierarchical reflexion analysis was introduced by Koschke et al. [101]). The authors extract a resource flow graph for an open-source instant messenger client and an Android instant messaging app for this purpose. A resource flow graph reflects the static entities of a program, such as classes, methods, and attributes, and the static dependencies, e.g., type dependencies, calls, or attribute usage. For the reflex-

ion analysis, the authors model the single access point security pattern. The model consists of the pattern's components and allowed dependencies. Next, the authors map the program elements to the architectural elements and the reflexion analysis searches for differences between the implementation and the pattern's model. Therefore, the analysis checks if the dependencies of the model and the implementation match. As a result, the reflexion analysis produces a list of convergent, divergent, and absent dependencies. An implementation-level dependency is convergent if there is a corresponding dependency in the architectural model. If no implementation-level dependency exists for an architectural dependency, an absent dependency entry is generated. Finally, if no matching architectural dependency can be found for an implementation-level dependency, it is a divergent dependency. In their case study, the authors then check if the structure of the implemented pattern matches the specified security pattern [46].

In 2020, Bunke et al. presented their approach to detecting security patterns based on connected object process graphs. Object process graphs (OPG) were introduced earlier as a means to model the existing interactions with an object [67]. OPGs were used to model static and dynamic interactions with an object. A connected object process graph is, according to Bunke et al., the connection of different object process graphs to show the interactions of different objects. They use their approach to identify instances of three different security patterns, namely authentication enforcer, information obscurity, and secure pipe, in 25 different Android apps. In total, they detected 12 out of 15 instances of the authentication enforcer pattern, 65 out of 87 instances of the information obscurity pattern, and 122 out of 146 instances of the secure pipe pattern [47].

4.5. Chapter Summary

This chapter showed an overview of static analysis. After introducing compiler construction, which explained the essential parts of a compiler used in static analyses, this chapter introduced different static analysis techniques. The section on static analysis techniques sketched widely used representation, such as control flow graphs and specialised methods, such as inter-procedural dataflow analysis, relevant to the thesis' remainder. Finally, the chapter introduces software architecture recovery based on a taxonomy proposed by Ducasse and Pollet.

Static analysis helps to extract information of software's given source code. Depending on the used techniques, it helps to identify dataflows or searches for implemented patterns. Software architecture recovery additionally helps to extract the software's architecture from its source code. All of these techniques

might be helpful when trying to search for architectural security flaws.

Similar, the techniques presented in the area of security pattern detection are helpful for detecting architectural security flaws since they are related to security patterns as the previous chapter showed. Nevertheless, the existing techniques focus on detecting the usage of security patterns, but do not focus on extracting more abstract information, such as the implemented authorisation policy.

Part II

Theme

Chapter 5

Research Topic

After the thesis' Prelude has given an overview of the topics of software security, software architecture and static analysis, this chapter will define the research area of this dissertation. The following Section 5.1 summarises those fundamental findings and relates them to one another in a problem description. Based on the fundamentals, Section 5.2 then describes the research questions that arise from this problem description. Section 5.3 sketches a possible problem solution, which the following chapters will explain in more detail. Finally, Section 5.4 summarises the results of this chapter.

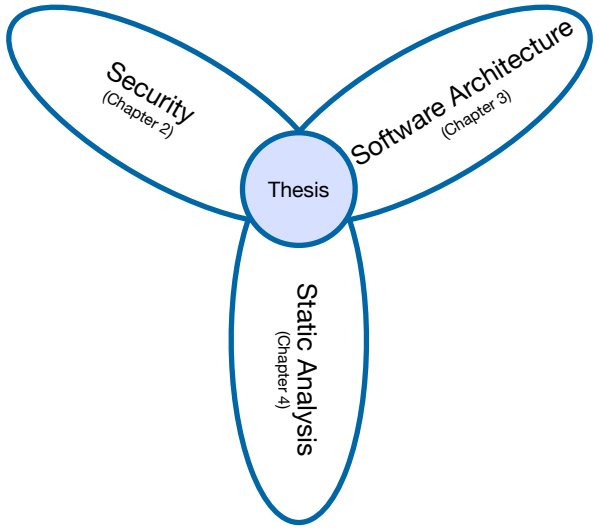
5.1. Problem Description

Figure 5.1 shows the different background chapters of the Prelude. In contrast to Figure 1.3, the various subject areas do not visually overlap here. In the following, this section brings the contents of the individual chapters together to motivate the thesis' research area, which is related to all the introduced topics.

As the chapter on security has shown, attacks on software systems have shifted over the years. In the beginning, many attacks were carried out at the network level. With the increasing spread of protection mechanisms at this level, such as firewalls and application proxies, attackers have sought new targets to attack a large number of applications. The attacks that followed focused on common implementation errors such as buffer overflows, SQL injections or command injections. With the increase in these types of attacks, countermeasures were rapidly developed, such as software libraries that were secure against this type of attack or static source-code analyses that could reveal such errors in existing software. This form of vulnerability still exists today. Nevertheless, a noticeable trend is that attacks are now becoming more application-specific and increasingly exploit conceptual weaknesses in these specific applications [83].

The software architecture of an application is the conceptual floor plan of software and is a means of communication to provide information about (planned) software. Good software architecture must, therefore, also make statements

Figure 5.1. Overview of the related research topics



about the security mechanisms of a software system. Depending on the architecture description framework used, there are different ways in which security can become part of an architecture: either by integrating the information on security into the existing architecture views or introducing views that explicitly deal with security.

None of the architecture description frameworks described which security aspects are relevant at the architecture level. Necessarily, a turn to information security is in order to approach this question. Information security states that the appropriate security objectives to ensure data's and systems' security are authenticity, integrity, confidentiality, availability, accountability, and anonymisation. To adequately address software security, these objectives must already be dealt with at the architectural level. In addition to potential security problems, information security also describes possible countermeasures. The most important ones here are authentication, authorisation, and cryptography, all of which help make an application secure.

Information security answers the question of which relevant security properties must be taken into account when planning software and how, in principle, these properties can be enforced. However, the software architecture can only ever be a model of the software. Thus, it is necessary to ensure that implementation and model correspond to one another. Microsoft's threat modelling supports the development process by describing the tasks to be fulfilled in the different software development phases, such as planning, implementation,

testing and release, to obtain a secure software system as an end product. A part of Microsoft's Secure Development Lifecycle that deals with architectural security is the presented threat modelling. First of all, it supports identifying critical resources, and then, with the help of the STRIDE approach, it systematically identifies possible dangers for the protection goals of information security. Nevertheless, there are still different aspects of threat modelling that need improvement. Threat modelling works on the software's architecture, and therefore on the software's model, it does not check the software's implementation. Conclusively, the extraction of architectural security views for a given software system and automatic detection of security flaws would be beneficial tasks, as Shostack mentioned [143]. These improvements compose the second and third interesting research areas shown in Table 5.1.

Table 5.1. Research areas

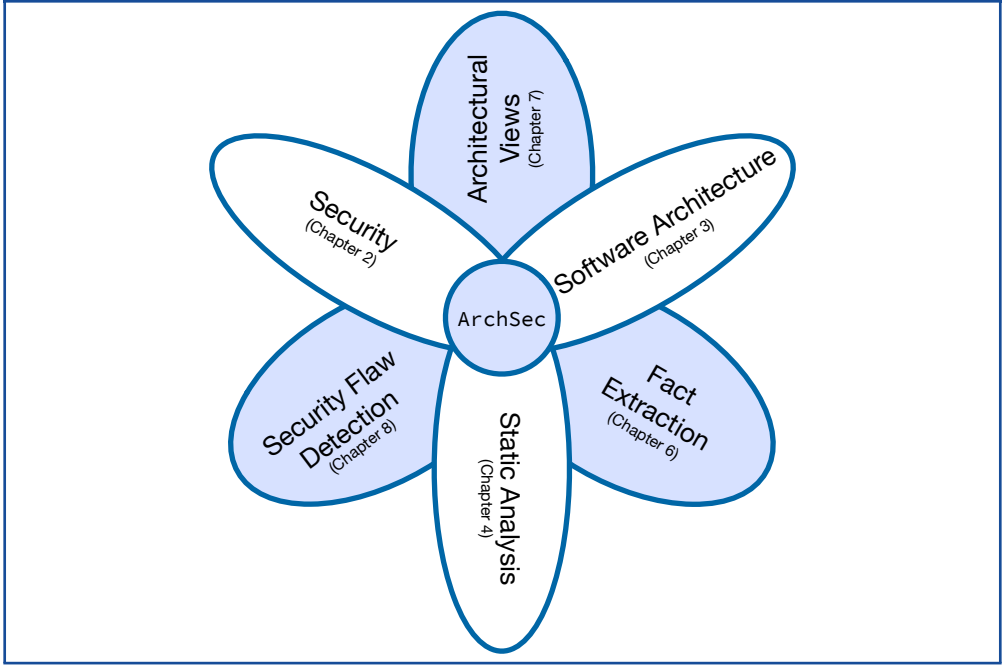
Name	Topic
T1	*More expressive dataflow diagrams.*
T2	*Automatic extraction of architectural security views.*
T3	*Automatic detection of security flaws.*

The last paragraphs discussed software architecture and its relation to security. So far, these paragraphs did not discuss the actual modelling of software architectures. The chapter on software architectures introduced different modelling techniques to represent various aspects of a software system. They differ in their level of abstraction, their modelling scope, and their degree of formalism. Security experts make use of all the presented modelling techniques to express different security aspects. The threat modelling approach mentioned above uses a relatively informal modelling approach by employing dataflow diagrams. The reason for using an informal model, according to various sources, is the ease of utilising it and its flexibility to adapt to most existing software and security aspects. Threat Modelling can even be done on a whiteboard, allowing one easily to discuss and change the dataflow diagram and the threat model. More formal approaches do not offer these advantages. However, other researchers wish for more expressive dataflow diagrams with better tool support, simplifying threat modelling and enabling automated security flaw detection [63]. Therefore, more expressive dataflow diagrams are an interesting research area and are also shown in Table 5.1.

5.2. Research Questions

In total, the problem description results in three research areas that this thesis delves into, as Figure 5.2 depicts. The light blue areas show this thesis's different contribution areas related to the research areas from the last section. The included references show which chapter details the contribution in this particular area. The combination of all these contributions resulted in ArchSec, the ARCHitectural SECurity tool suite [39].

Figure 5.2. Overview of the approach's contribution



The first research area, $T1$ from the previous section, combines software security and software architecture topics, deals with security-related architectural views and lays the foundation for other research areas described in the last section. The first research question in this area is: *RQ1—How can dataflow diagrams—used in threat modelling—be enhanced to support a more formal expressiveness?* As described, more formal dataflow diagrams are the pillar stone for automated threat modelling. The conditions for a more formal version of dataflow diagrams are slightly contradictory. The dataflow diagrams have to be more expressive while still be easy to use. Besides a more formal presentation, the information to present comprises the second research question: *RQ2—What*

kind of architectural information and security information is of interest during threat modelling? This research question describes the aspects that dataflow diagrams should be able to model.

Primarily, the second question results in a list of requirements for the research area $T2$, the automatic architecture view and security fact extraction. In particular, this area utilises state-of-the-research static source-code analyses to extract dataflow diagrams. This research area has to deal with two research questions as well, to do so. The first question deals with architecture recovery: *RQ3—What kind of relevant architectural information can be extracted automatically to create dataflow diagrams for existing software systems?* While this research question focuses on extracting the software's floorplan, it does not necessarily identify security measures that are in place. Therefore, the other research question of this area is: *RQ4—Which architectural security information can be automatically extracted using static software analysis?* This extracted information then becomes part of the previously created security views. Information security and the STRIDE approach give us a principal direction for this research question.

The last research area, namely $T3$, deals with automatic security flaw detection. Therefore, it has to investigate how the detection of threats and possible mitigation can be automated using extracted dataflow diagrams. Additionally, the detection should also work for manually created architectural views, for instance, when these are made while planning new software systems or features. In essence, we can formulate the first research question in this research area: *RQ5—How can we describe architectural security flaws for dataflow diagrams?* A security flaw description consists of different parts, but the most important ones are the notation of security anti-patterns. A security anti-pattern here is a part of the dataflow diagram that might be a security problem. Another important aspect is the notation of security patterns, a part of the dataflow diagram that secures a previously identified security flaw. The second, tightly coupled research question in this research area is: *RQ6—How can instances of the architectural security flaw descriptions be automatically identified?* This question is important since the notation of rules is worth less if there is no automated way to check them.

5.3. Approach

The ArchSec approach has been designed and implemented to cover the research areas and investigate the research questions described in the last section. The approach is based on the principles of model-driven software de-

velopment. In model-driven software development, the domain's data are the focus of development and a model of them is created first. The model is instanciated at runtime. This can be done programmatically, with the help of domain-specific languages, or by visual modelling. In addition, there are domain-specific languages for describing how to parse a text into a model instance (text-to-model transformation), transforming data between different models (model-to-model transformation), and generating textual representations for a given model (model-to-text transformation).

Using static analysis, ArchSec extracts different architectural aspects and security facts. Since static analysis is a language-dependent task, it has been implemented for Java-based software systems. Essentially, it supports different frameworks, such as JavaEE and Android. This intermediate model is then translated into extended dataflow diagrams using a model-to-model transformation. A diagram editor can visualise the extracted diagrams or allows the manual creation of a new diagram. ArchSec defines an extensible extended dataflow diagram that adds additional node types, edge types and attributes. A diagram can automatically add security requirements and mitigations based on the used node and edge types. Furthermore, an extended dataflow diagram can be checked using a predefined knowledge base or against self-written rules.

A knowledge-base rule consists of matching conditions that identify a possible threat. Additionally, a rule has a detailed description of the threat and a rough estimation of the threat's severity and likelihood. Furthermore, there are mitigation conditions that check if there is appropriate mitigation in place. An automatic security flaw check results in an automatically created threat model that lists possible security flaws and identified mitigations. The threat model uses the STRIDE approach to categorise the identified threats.

The knowledge base allows specifying rules to identify general security flaws relevant to a large number of applications. Nevertheless, not all security flaws are application-independent. Here two possible solutions apply. The first option is to create application-specific knowledge bases that capture the security aspects relevant to an application. The second option is to generate application-specific rules based on a specification. For such rules, e.g., authorisation rules, special editors have been developed to allow an easy specification of the authorisation policy. The editor transforms the specification into a custom knowledge base applicable to an extended dataflow diagram.

The implementation of ArchSec uses two main frameworks for its realisation. For the static analyses, it uses Soot, a static analysis and manipulation framework for Java-based systems [162]. Since it uses Java bytecode as its input, Soot allows analysing software systems without having access to its source code. The diagram editor, policy editor, knowledge-base editor and the extended dataflow

checker integrate into the Eclipse application framework. The diagram editor, policy editor, knowledge-base editor, and the extended dataflow checker are integrated into the Eclipse application framework. The integration allows creating a threat modelling environment where all features mentioned above can be used together.

5.4. Chapter Summary

This chapter summarised the background chapters on security, software architecture, and static analysis. The information from these chapters led to the description of research areas the remainder of this thesis sought into. The research areas resulted in six general research questions that the following chapters attempt to answer. Table 5.2 summarises these questions to give an overview of what questions Theme (part II) and Variations (part III) will answer. Theme will focus on detailing the approach proposed in this chapter, and Variations then evaluates it.

Table 5.2. Thesis' general research questions

Name	*Topic*	*Question*	*Section*
RQ1	T1	*How can dataflow diagrams be enhanced to support a more formal expressiveness?*	7.1
RQ2	T1	*What kind of architectural information and security information is of interest during threat modelling?*	6.1
RQ3	T2	*What kind of architectural information can be automatically extracted for existing software systems?*	6.1
RQ4	T2	*Which architectural security information can be automatically extracted using static software analysis?*	6.1
RQ5	T3	*How can we describe architectural security flaws for dataflow diagrams?*	8.2
RQ6	T3	*How can instances of the architectural security flaw descriptions be automatically identified?*	8.4

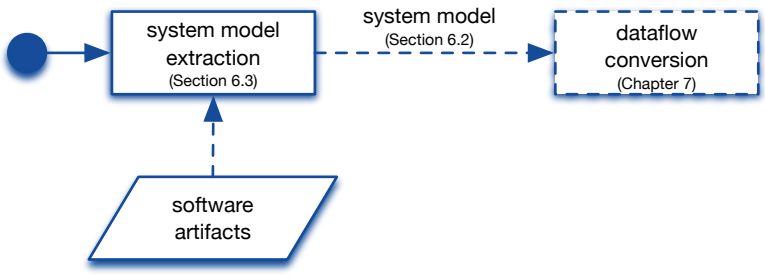
Chapter 6

System Model Extraction

To discuss and answer *"RQ1—How can dataflow diagrams be enhanced to support a more formal expressiveness?"* it is necessary to know which information needs to be stored in dataflow diagrams since the results of this assessment can influence the formalisms to choose. Therefore, Section 6.1 recapitulates the introductory chapters' content and deals with the question: *"RQ2—What kind of architectural views and security information is of interest during threat modelling?"*. The resulting list of analysis requirements must then be examined to determine which of them can be extracted with static analysis. Therefore, Section 6.1 discusses two additional questions: *"RQ3—What kind of architectural information can be automatically extracted for existing software systems?"* and *"RQ4—Which architectural security information can be automatically extracted using static software analysis?"*. Finally, this chapter explains the system model's static extraction and describes the design goal of the extraction process, while the next chapter details the translation into dataflow diagrams.

The extraction process aims to extract architectural and security-relevant information from an already existing software system. Therefore, it has to carry out an architecture recovery and detect coding patterns that represent security measures discussed in the first two sections of this chapter. Figure 6.1 shows the outline of this process.

Figure 6.1. Outline of the system model's extraction



The first step of the ArchSec approach—described in Section 6.3—extracts a system model using different static analysis techniques. The system model is an intermediate representation containing static information on the software system's structure and information on employed security mechanisms. However, in the beginning, Section 6.2 introduces the system model's core parts before the extraction is presented. The extraction process should support different Java-based frameworks and security mechanisms, especially those identified in Section 6.1. Therefore, the system model and the extraction process must be extensible to add new features.

6.1. Analysis Requirements

The first research question to take a look at in order to identify the analysis requirements is: *"RQ2—What kind of architectural information and security information is of interest during threat modelling?"* The introductory chapters have already mentioned some answers to this question, which this section condenses in its beginning to a list of requirements.

Threat modelling, as described in Section 2.3.2, decomposes a system into hierarchical *components*. Then, it identifies the components' *assets* and *entry points*. Finally, it searches for *dataflows* within the software system and *interactions* with external systems. All of this architectural information is of interest to the creation of dataflow diagrams. Therefore, it is desirable to extract all of these aspects. An obvious tool of choice is static analysis since it supports the automation of this extraction. Additionally, software architecture recovery deals with identifying architectural components within a given implementation and, for this reason, can be helpful here.

The security aspects relevant to threat modelling vary widely. The section on information security explained the importance of the topics *authentication*, *authorisation*, and *cryptography*, which are means to mitigate attacks on information security protection goals that have been turned into security patterns. Besides, threat modelling can deal with other security issues, such as identifying *insecure execution of code* and *application-specific security risks*, which can indirectly result in threats to the mentioned goals. Pattern detection, in turn, is used in software architecture recovery to identify architecture-related artefacts. Therefore, the detection of security patterns goes perfectly along with the software architecture recovery.

After dealing with the necessary information for creating dataflow diagrams, research question 3 studies the possibility of automatic extraction of this information: *"RQ3—What kind of architectural information can be automatically*

extracted for existing software systems?" Due to the growing complexity of today's applications, many software frameworks, such as JavaEE and Android, rely on component-based development. The frameworks provide a certain number of different components, such as activities, services, content providers, in the case of Android, or managed beans, web service end-points in the case of JavaEE. The frameworks usually deal with the creation of these components and their lifecycle management. Whilst this allows faster development of applications, these aspects complicate static analysis in many cases. Nevertheless, these components offer a good starting point for software architecture reconstruction based on these framework-specific mechanisms since they allow software architecture recovery to identify components and their entry points. Identifying assets is a more complicated task since they depend on the concrete application's purpose. Information, seen as assets in one application, might not be assets in other applications. Therefore, automatic detection of assets is challenging. Data flow analysis usage allows identifying the flow of assets from the identified entry points or within a software system. Finally, with knowledge of the framework APIs' semantics, it is possible to identify method calls corresponding to external systems' interactions. If there are dataflows of assets that end in such methods, it results in a flow of that asset out of the system's control.

Finally, the last research question to focus on is *"RQ4—Which architectural security information can be automatically extracted using static software analysis?"* A software's implementation potentially contains all mentioned security mechanisms, such as authentication, authorisation, and cryptography and might even use one of the corresponding security patterns introduced in Section 3.4. It might be the case that the frameworks adapt the patterns to meet the frameworks' unique requirements. Since the security mechanisms are part of the implementation, one can conclude that they can be extracted using static analysis. Nevertheless, some security aspects are runtime-dependent, such as the user- and permission-assignment in role-based access control (cf. Section 2.2.4). A custom static analysis could extract this information for a concrete system, but it is not the dissertation's subject.

An important fact to keep in mind is that there is a difference between analysing a component framework (sometimes referred to as a component container) and analysing its usage. The same applies to the implementation and usage of patterns, for instance, the authorisation pattern. It is consequently necessary to review both the implementation and usage to ensure an application's security. This thesis focuses on both framework and pattern usage since the frameworks and patterns are used by many applications allowing the automation of their usage analysis. This thesis assumes the frameworks implement their security

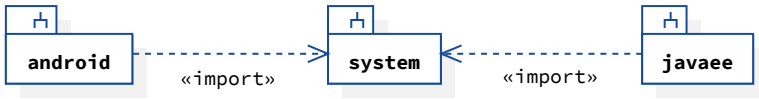
measures correctly and concentrates on their usage. The correctness of application containers, frameworks, and patterns need an additional review to ensure the correct implementation, but it is out of the scope of this thesis. In the context of this work, it is assumed that this has already happened due to the widespread use of these frameworks and that any errors in their implementation have already been discovered.

The usage of static analyses sets limits to the results of the approach. For instance, it might be possible to identify that a resource is opened in the software's implementation, but since the filename might depend on some external input, it is impossible to determine the concrete accessed resource. One can argue to augment the results of the extraction process with the result of dynamic analyses to have a more specific picture of the program's behaviour. The ArchSec approach does not follow this idea since, from the security point of view, it is only important to know that an external system or user can determine the opened file since this is a possible attack vector. The Common Weakness Enumeration, which collects and categorises weaknesses in today's software systems, lists this kind of problem in their list of the 25 most dangerous software weaknesses. The corresponding entry at rank 12 is: *"CWE-22: Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')"*[1].

6.2. System Model

This section explains the internal system model in greater detail. The system model is an extensible data model for describing software systems and serves as ArchSec's analysis intermediate representation. The `system` subsystem describes the heart of the model. It is extended by analysis-specific, framework-specific or domain-specific packages. Figure 6.2 shows a top-level package diagram of currently existing extensions.

Figure 6.2. Package diagram of all existing subsystems



The complete model consists of three subsystems. In the beginning, Section 6.2.1 details the central `system` subsystem. It contains framework-agnostic parts of the model and provides a fundamental core structure. This section, in contrast to the following ones, gives a very detailed model description. The

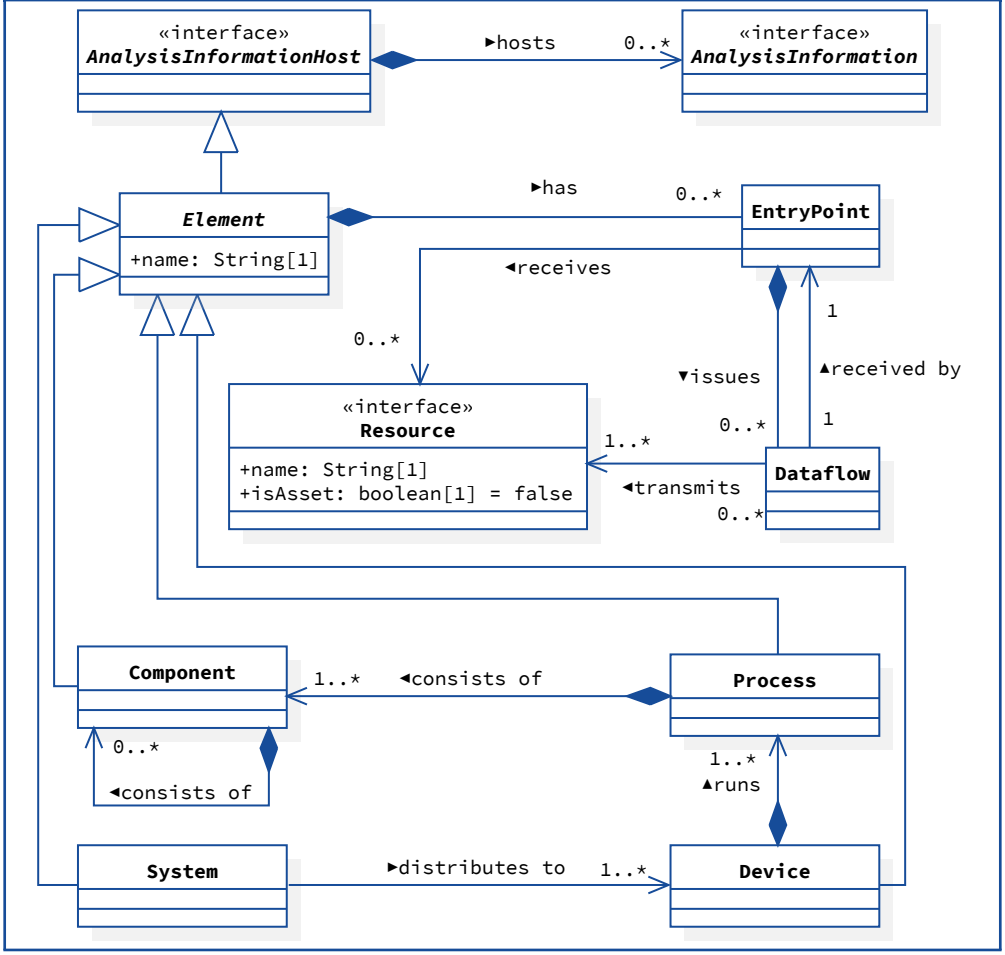
[1]The complete entry can be found https://cwe.mitre.org/data/definitions/22.html

remaining two subsystems import `system` but do not access each other. Namely, the `javaee` and the `android` subsystems are containing JavaEE-specific and Android-specific extensions. A detailed description of these subsystems is not part of this dissertation and is therefore beyond its scope.

6.2.1. Subsystem `system`

The subsystem `system` contains elements to describe software systems without focussing on specific applications or frameworks, besides the Java programming language. Figure 6.3 shows the package hierarchy of the subsystem. The following subsections delve into the depicted packages in more detail.

Figure 6.3. Package diagram of the `system` subsystem



The `core` package is at the centre of this subsystem and ArchSec's data model. It provides an extensible foundation for modelling software systems and reflects many of the architectural requirements. Additionally, the `filesystem` package uses the `core` package to add basic support for files and directories. The `callgraph` package adds a context-sensitive callgraph to the subsystem to model a program's static call relations. Finally, the `java` package enhances the model with elements for Java's static type information.

`core` Package

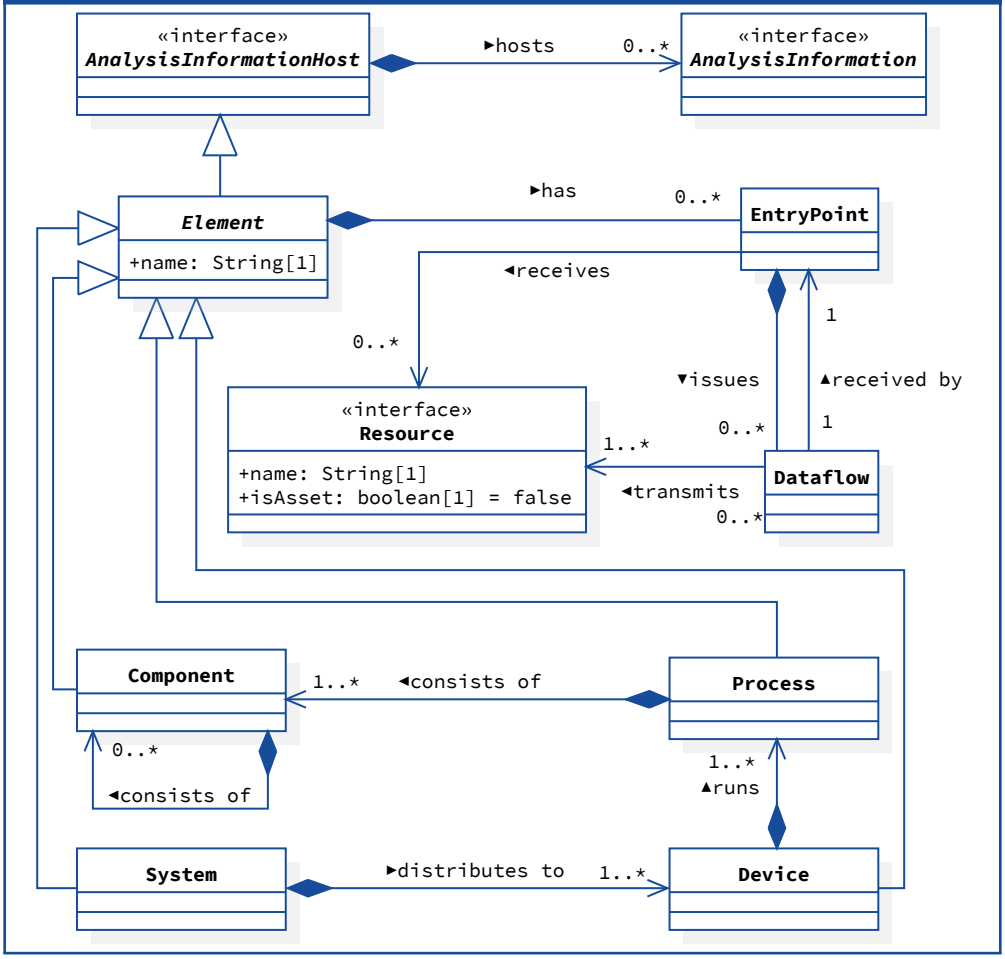
The `core` package contains means to describe a basic software system and is extended by domain-specific or analysis-specific packages later on. The package models the architecture-related requirements listed in Section 6.1. Figure 6.4 shows the package's content.

The root element of the data model is a `System` that distributes to an arbitrary number of devices. A `Device` is a hardware component, for instance, a personal computer, a smartphone, or a server, that hosts a part of the system. Furthermore, a device can be part of the software's deployment environment, such as a router or a hardware firewall. Dynamic network analyses, for example, are capable of adding this information but fall out of scope due to the static nature of the analyses presented in this thesis. A `Device` may run up to several processes implemented in any existing programming language. Each `Process`, in turn, consists of several software `Component`s which can be structured hierarchically.

Each of the so far mentioned classes extends the class `Element`. An element corresponds to a component mentioned in the analysis requirements. An element has an arbitrary number of `EntryPoint`s that receive resources. Entry points correspond to a system's element that is accessible by others. A resource has a name and might be an `Asset` according to the threat modelling's meaning. There are many different possible resources for a system, such as local files (see the following subsection), documents loaded from remote locations, or memory locations. An entry point can issue `Dataflow`s to receiving entry points. The dataflow may transmit resources to other entry points.

There are two ways for other packages to extend the `core` package. On the one side, a class may extend one of the mentioned classes to define domain-specific devices, process types, or components. On the other side, a class may inherit from the `AnalysisInformation` interface. Every class implementing the affiliated `AnalysisInformationHost` interface can host an arbitrary amount of analysis information. This way, supplementary analysis information can be added to any model element without changing the base model. Within the `core`

Figure 6.4. Class diagram of the `core` package



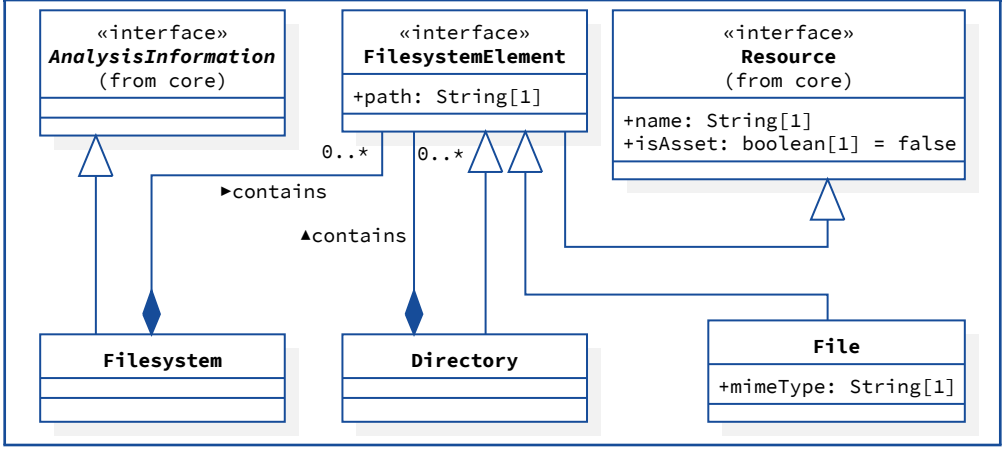
package, all non-abstract classes inherit from `AnalysisInformationHost`. The remainder of this chapter shows exemplary usages of both extension mechanisms.

`filesystem` Package

Most computing devices contain persistent memory storing directories and files that may be accessed by processes for configuration purposes, data input, or data output. The `filesystem` package adds support for these elements to the `core` model. It contains a class named `Filesystem` that implements the `AnalysisInformation` interface. Therefore, any `AnalysisInformationHost` can store a filesystem. An associated OCL invariant only allows a `Filesystem`

to be assigned to model elements inheriting from `Device`. Figure 6.5 shows all elements of the `filesystem` package.

Figure 6.5. Class diagram of the `filesystem` package



A `Filesystem` hosts an arbitrary number of `FilesystemElements`, which is an abstract model element. There are two concrete file system elements, namely a `Directory` or a `File`. Since these three classes implement the composite pattern (please compare Gamma et al. [77]), a `Directory` can contain an arbitrary number of `FilesystemElements` as its children. The composite pattern allows creating an unlimited nested directory hierarchy with files in all directories. The inherited attribute `name` of a `FilesystemElement` stores the name of directories and files. Furthermore, files have a `mimeType` attribute that specifies the content's type of a specific file.

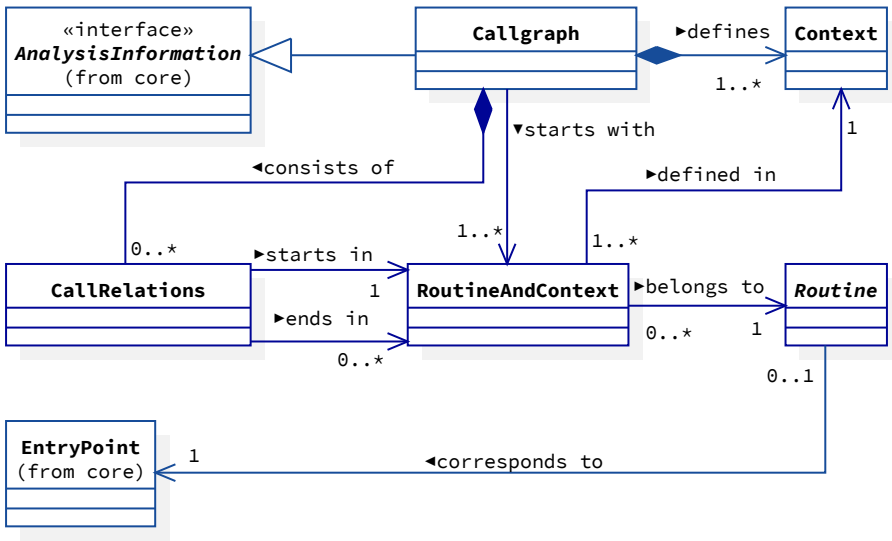
Since the `FilesystemElement` implements the composite pattern, adding files containing other files, such as tar-archives or compressed files, is possible. This aspect is relevant for Java-based systems since they are usually deployed using compressed files. The program can then read files contained in the compressed file, for instance, for configuration purposes.

`callgraph` Package

Many, primarily inter-procedural, static analyses employ a callgraph to perform their tasks. This package adds a language-independent context-sensitive callgraph to the subsystem `system`. Figure 6.6 shows the package's class diagram.

A `Callgraph` defines `Contexts` to describe different levels of preciseness. Since many modern application frameworks, such as Android or JavaEE, do not

Figure 6.6. Class diagram of the `callgraph` package



have traditional main methods, the callgraph starts with multiple entries. An entry is of type `RoutineAndContext`. This type is defined in a specific `Context` and belongs to a `Routine`. A routine may correspond to an `EntryPoint` from the `core` package. The `RoutineAndContext` can be interpreted as a routine that is executed in a particular context. The `Callgraph` furthermore contains `CallRelations`. Each call relation starts in a `RoutineAndContext` and ends in one. If the calculated callgraph is context-insensitive, it can be stored as well by using a single context.

The software architecture recovery process is going to split the analysed software into several components. Since the `Callgraph` is an `AnalysisInformation`, it is possible to append a partial callgraph to each architectural component. Partial means that the whole-application callgraph is assigned to a component that starts at the component's entry points and does not belong to other components. A partial callgraph enables an assignment of security findings to architectural components.

`java` Package

A widely-used and by ArchSec supported programming language is the Java programming language. Consequently, the subsystem `system` contains a package that models Java type information. The `java` package allows adding static type information for Java-based programs. Hence, it contains means to model

existing packages, types, fields, and methods. Figure 6.7 depicts the package's content. Please note, the diagram omits many attributes to ensure better readability.

The package's root class is the class `JavaTypeInformation` which inherits from `AnalysisInformation` and contains a reference to a single root package. A `JavaPackage`, in turn, has a `name` and contains an arbitrary number of `ReferenceType`s and child packages. The contained child packages allow one to model Java's package hierarchy. Java's type system starts with the base class `Type`. A type can model Java's base types, such as `int`, `char`, or `boolean`. The subclass `ReferenceType` contains `JavaField`s and `JavaMethod`s and is the base class for `JavaClass`es, `JavaInterface`s, and `JavaArray`s. A `JavaArray` is of a specific type and might have a dimension. `JavaMethods` declare formal parameters.

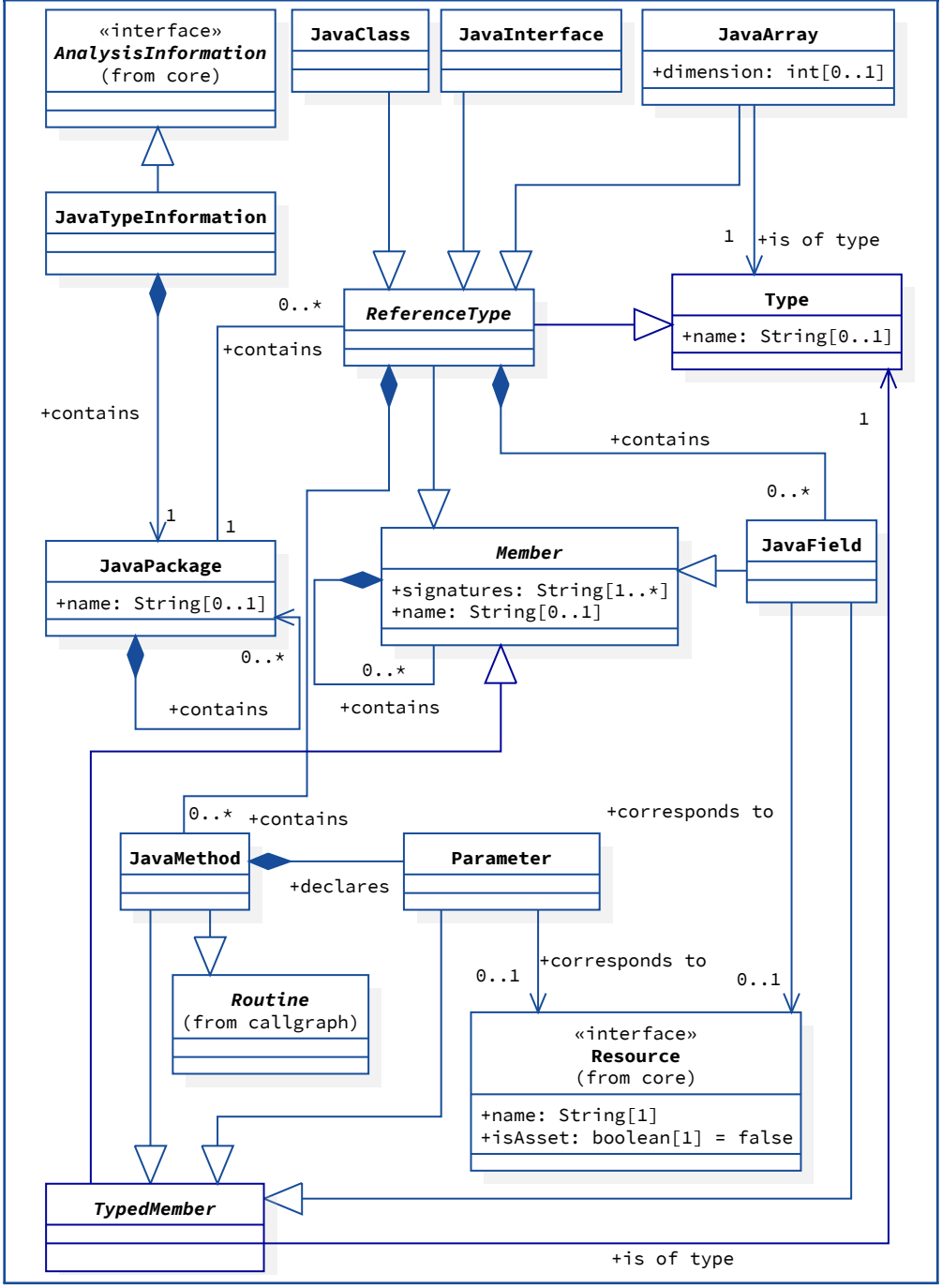
Please note that `JavaMethod`s extend `Routine` from the `callgraph` package allowing to specify callgraphs for Java-based systems. Furthermore, `Parameter`s and `JavaField`s inherit from the `core` package's `Resource` class and can be assets in the threat modelling's sense.

6.2.2. Subsystem `android`

Android is a Java-based middleware and operating system mainly known for its use on mobile devices, such as mobile phones and tablets, but Android also runs on car's media systems, smart televisions, and netbooks. The basic operating system works with a customised Linux kernel tailored to support a strict separation of user processes. Android's middleware is a Java-based collection of framework APIs and services supporting various communication mechanisms, storage possibilities, and sensitive information. Since we use smartphones in our daily lives very frequently, they store personal and sensitive data, making security and privacy an essential topic. Consequently, Android has different built-in mechanisms supporting security.

As mentioned, Android supports a strict separation of different processes. Each installed application (called App) runs under its user, disallowing other processes to access the application and its data using standard file operations. Android supports an individual mechanism of inter-process communication and remote method invocation, the Android binder for applications to interact. The Android binder is a kernel device allowing interaction between processes without sharing memory. Each process has its binder device, where a process copies its request to. The operating system then copies the data to the called process' binder, where the request is read from the target process. After processing the request, the target process can send back a reply to the calling process.

Figure 6.7. Class diagram of the `java` package



The programming model of Android Apps differs from classical JavaSE applications. In contrast to JavaSE applications, an Android App does not have a main method. Android is a component-based framework and supports four distinct types of components. The most prominent ones to the end-user are *activities*. The sum of all activities makes up the application's user interface. The main idea is that there is an activity for every group of information to display and that the application navigates between these activities. A music player app, for instance, has an activity for the list of all stored music files and a particular activity for a song's details. If the user

While an App's activities deal with user interaction, they should not contain any long-running calculations. If an activity contains a long-running algorithm, the application will freeze from the user's point of view. Since freezing is harmful to the user experience, such algorithms need to be outsourced to service components. A frequently used example of services is a playback service that plays music in the background. Via an activity, the user can choose the music to play and pause or stop the music, while the actual playback is outsourced to a service.

Services allow bi-directional communication between two applications. A disadvantage of this approach is that the client app needs to know the service at compile time. There might be situations where an app wants to inform any other App about an event without knowing it, such as informing all Apps about an established connection to the worldwide web. For receiving such information, Android offers the broadcast receiver component. A *broadcast receiver* registers for specific intents (the Android terminology for an event) and gets called whenever this kind of intent is broadcast. There is even a concept for permission delegation called *pending intents*. The sending app can pass a permission to the receiving app, such as reading a file that belongs to the sending app. This way, the playback service would be able to play a song that belongs to a different app.

Finally, the *content provider* component provides a standard interface for storing and reading information, for instance, from an application-hosted database. The content provider stores information that, for instance, is shown in the user interface and processed by a service. This information may be sensitive. In the media player's case, it might store meta-information about the tracks or playing statistics.

Each of the introduced component types has its specific lifecycle that is steered by the Android framework. It deals with creating the components as well as invoking and eliminating them. Developers can decide to export each component. An exported component is accessible to other apps. Consequently, every exported component is an entry point to the app and part of its attack

surface.

The `android` subsystem extends the central subsystem with Android-specific elements. It specifies new elements for the introduced Android components that inherit from the core package's `Component`. If components are exported, they automatically correspond to entry points. Furthermore, the subsystem introduces additional Android-specific `AnalysisInformation`.

6.2.3. Subsystem `javaee`

The Java Platform, Enterprise Edition [58], nowadays known by Jakarta EE [155], is based on the Java Platform, Standard Edition and standardises a development and runtime environment for transaction-based web applications. It aims to provide design and architecture patterns to make the development of such applications more straightforward. This document's remainder uses the term JavaEE for any Java Platform, Enterprise Edition, or Jakarta EE version.

Java's community regularly defines APIs relevant to the Java community based on Java Specification Requests (or shortly JSR). For example, the Java Persistence API in version 2.2 is the result of Java Specification Request 338[57]. It defines an API for automatically persisting instances of annotated classes to SQL-based databases. Different JPA providers provide an implementation of JSR-338. The basic idea is that the various providers are interchangeable. Each JavaEE version bundles a set of JSRs to specify a well-defined development and runtime environment for applications. JavaEE applications have specific deployment formats and are not directly executed using a Java virtual machine. Instead, they are deployed to an application container providing implementations for all required APIs. Like the JPA providers mentioned above, there are various application containers. Some of them are free of charge open-source containers, whilst others are proprietary ones. Besides the standard APIs, there are different additional libraries to overcome the shortcomings of the JavaEE versions.

JavaEE supports different kinds of components that may serve as entry points to a JavaEE system. With Java ServerPages and Java ServerFaces, there are two technologies for generating dynamic web pages whose components are accessible by standard web browsers. Stateful and stateless session beans are long-living server-side components that are also accessible from other applications. These specifications define different kinds of components and entry points that need to be supported by ArchSec's intermediate representation.

JavaEE included support of security-relevant aspects from its first version. The community enhanced security support over the years, according to the community's needs. The current JavaEE version requires application containers

to implement JSR-375, the JavaEE security API. From the beginning, JavaEE supported transport encryption, different subjects and access control. The specifications call the access control "role-based access control". However, it is not a role-based access control model, according to Sandhu et al., introduced in Section 2.2.4. In JavaEE roles are assigned to users in the form of labels. These labels are then used to protect specific methods. In JavaEE, it is impossible to specify constraints for other elements, such as variables or attributes. Therefore, it merely resembles a simplified access matrix model introduced in Section 2.2.2.

6.2.4. Summary

This section introduced ArchSec's analysis intermediate representation using UML package and class diagrams. While it shows the central aspects of the model in greater detail, it gives only a brief summary of the extensions made for Android and JavaEE.

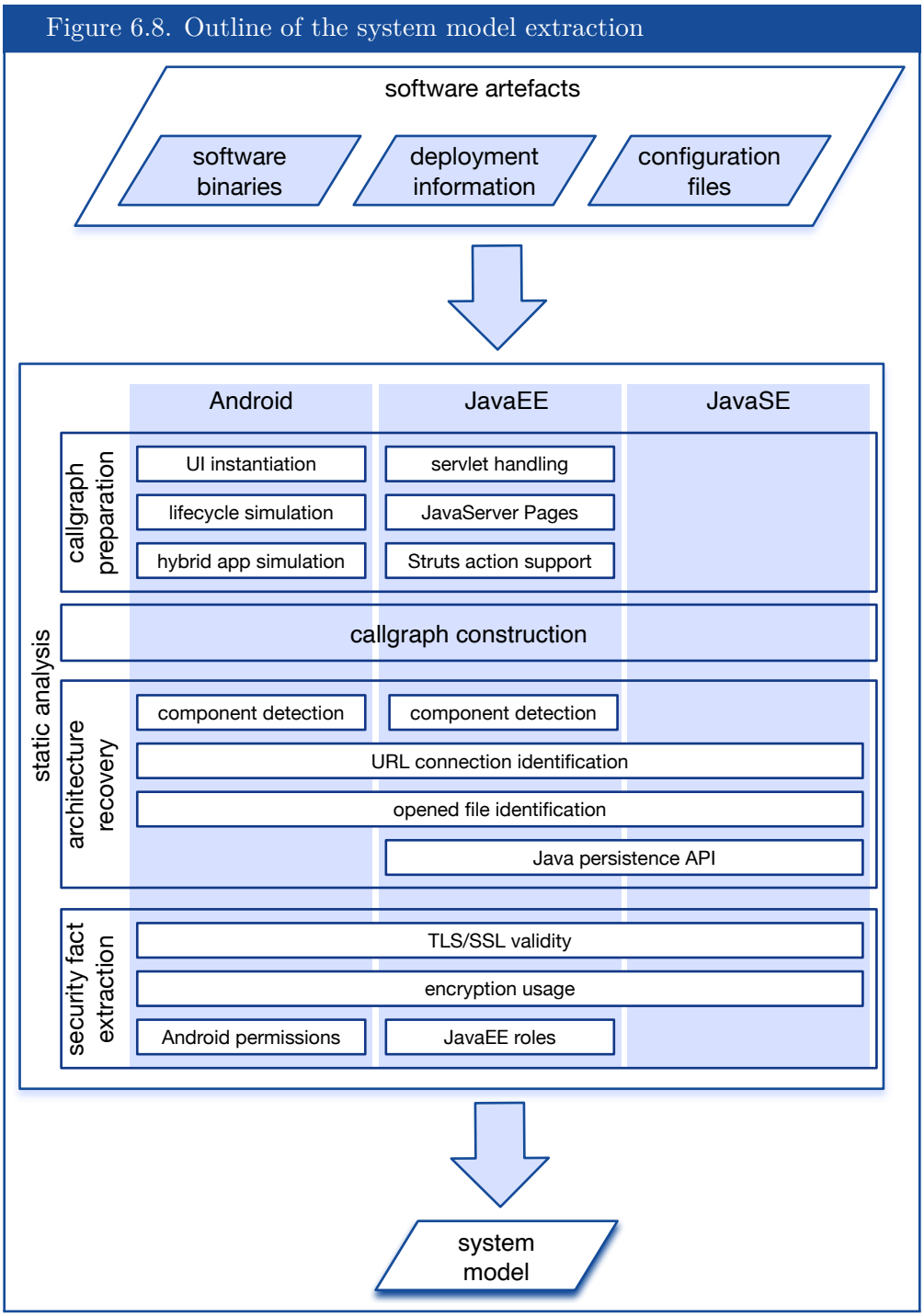
6.3. Extracting System Models

The application model extraction is programming language-, platform- and framework-dependent. The remainder of this section explains the extraction process and the reasons for its dependence. Figure 6.8 shows the outline of the approach. The extraction process extracts the system model based on different software artefacts related to the analysed software that poses as input, such as software binaries, deployment information, and configuration files.

The extraction process is divided into four different phases, which the following section details. First, there is the callgraph preparation phase. Its task is to extract information necessary to analyse applications without an explicit main method and add missing information regarding Java's heavy usage of program introspection. The second phase generates the callgraph used in the later phases. Afterwards, the third phase deals with extracting architecture-relevant information, while the last phase identifies security facts. All these phases make use of different widely-used static analyses, such as intra- and inter-procedural dataflow analysis, control-flow information, dominance information, and type information.

Each of the white boxes within Figure 6.8 represents an implemented static analysis enhancing the system model. Horizontal layers represent the introduced phases. A placeholder callgraph-layer is contained in the figure to give a better understanding of the analysis pipeline. The pipeline makes use of pre-made callgraph construction algorithms. The light blue swimlanes show the analysis

Figure 6.8. Outline of the system model extraction



steps' correspondence to the different application types used, such as Android, JavaEE, and JavaSE applications.

6.3.1. Callgraph Preparation

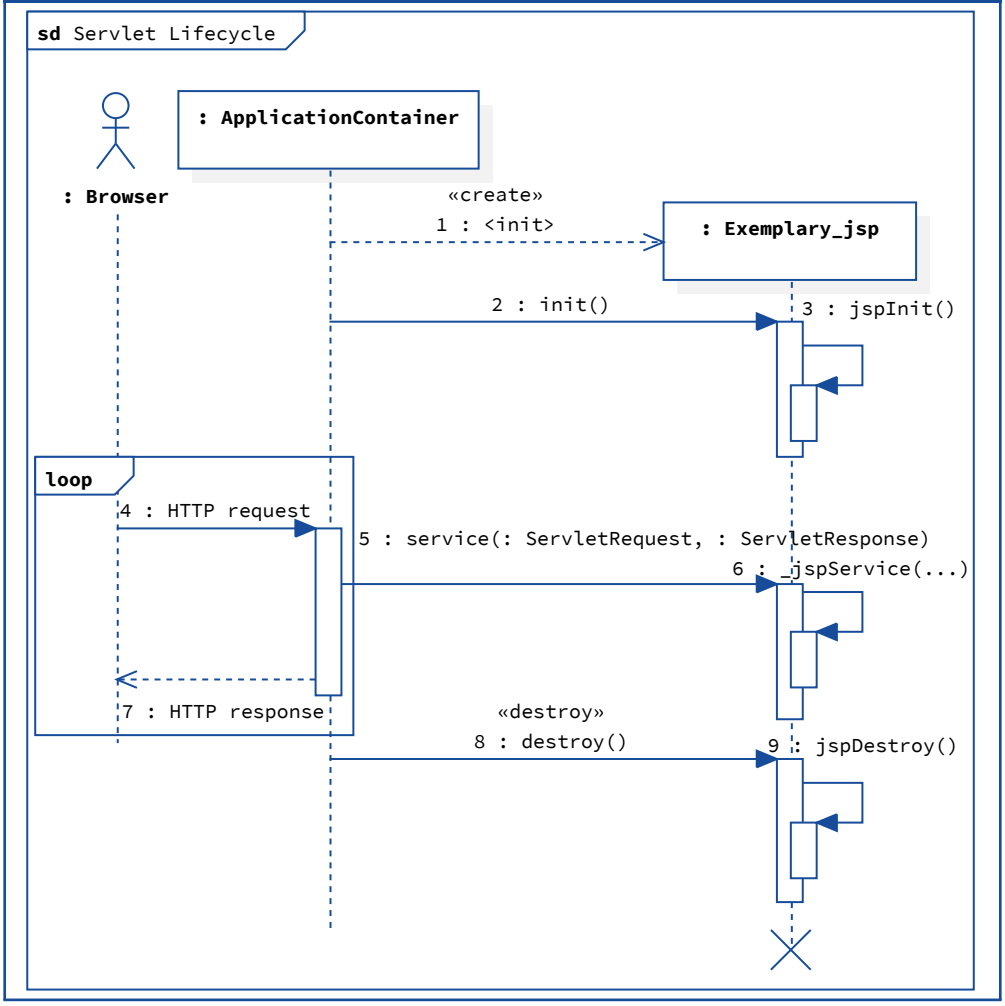
As mentioned in the sections on Android and JavaEE (cf. Section 6.2.2 and Section 6.2.3), these platforms' applications do not have a main method. Instead, the framework takes on the task of instantiating classes and calling them according to predefined life cycles. Nevertheless, the missing main method poses a challenge to the static analysis since most analyses, such as the callgraph algorithms, assume an existing main method. ArchSec uses static analysis to extract information on the application's structure and generates an artificial main method to overcome this problem. This artificial main method takes the framework-specific lifecycles into account to mimic the behaviour of the framework correctly. This section describes the steps of the callgraph preparation for a JSP-based JavaEE application.

For JavaEE-based software systems providing dynamic webpages, the web-archive deployment descriptor is parsed first. This file describes the mapping of HTTP requests to Java code. In a standard JavaEE application without additional libraries, developers can either implement a class extending the `Servlet` class, use a JSP mapping, or use a JSF processor. JSP and JSF are templating languages for generating dynamic web pages. While JSF uses XHTML for representation and a runtime interpretation, JSP files are translated into Java bytecode, using a JSP compiler, and then executed. The generated classes extend from the already mentioned `Servlet` class and are executed for incoming HTTP requests. The generated `_jspService` method creates a response document consisting of the static part of the template and the dynamically generated content.

Figure 6.9 depicts a UML sequence diagram describing the servlet's lifecycle. At runtime, the `ApplicationContainer` instantiates the servlet and calls `init`. The JSP servlet's base implementation then calls `jspInit`. This method allows a programmer to initialise the required resources of the servlet. Afterwards, the servlet is then called for processing HTTP requests. The `ApplicationContainer` calls `service`, which in turn calls `_jspService`. The service method processes a request and generates the response document. If the application container shuts down or discards the servlet for other reasons, the method `destroy` is called. The default implementation, once again, calls the JSP-specific method `jspDestroy`. This time, the programmer can discard reserved resources properly,

Listing 6.1 shows an exemplary JSP file. The JSP-Servlet prints all contents,

Figure 6.9. Lifecycle description of HTTP Servlets



except for escaped parts, of the file to the servlet's response. The character sequence `<%` starts an escaping while `%>` closes an escaping. The character after an escaping start defines where and how the content of that escaping is inlined. An exclamation mark tells the JSP compiler to copy the content into the class, while the equal sign tells the compiler to include the result of the code in the resulting response. The shown listing defines two methods, namely `jspInit` and `jspDestroy`, and therefore, overrides the default implementation of these methods. The remainder of the file consists of the template, which will be rendered on request. The file contains a simple HTML document. The template

Listing 6.1. Exemplary JSP file

```
<%! public void jspInit() {
   this.log("jspInit()");
 }

 public void jspDestroy() {
   this.log("jspDestroy()");
 }
%>

<html>
  <body>
    <h1>Hello <%= request.getParameter("name"); %></h1>
  </body>
</html>
```

includes the value of the HTTP request parameter `name` in its response.

Finally, Listing 6.2 shows the Java code of the processed JSP file. The generated class inherits from `HttpJspBase`, and it contains the two methods defined in the JSP file and a generated `_jspService` method. It writes the content of the JSP file to its response's output, including the result of the Java expression[2]. The error handling code generated by the JSP compiler is omitted here for a shorter Listing.

The callgraph preparation uses the specified lifecycle to generate an artificial main method. Therefore, it first extracts the information on existing JSP servlets and adds this information to ArchSec's data model. To do so, it uses a pattern-matching approach and looks for classes inheriting from `HttpJspBase`. Consequently, it expects the JSP files to be compiled by a JSP compiler. After detecting all JSP files, additional information is collected, such as existing filters, and added to the data model as well. Filters are JavaEE's realisation of the interceptor pattern [48]. The enriched system model is then used to generate the main method using model-to-text techniques. First, the central idea is to create all instances, initialise them, call the service methods, and properly destroy the JSP classes. Next, the entry-point generator creates Jimple code, the intermediate representation used by Soot which is then loaded into the currently running Soot analysis.

Figure 6.10 presents a UML sequence diagram of a generated main class with

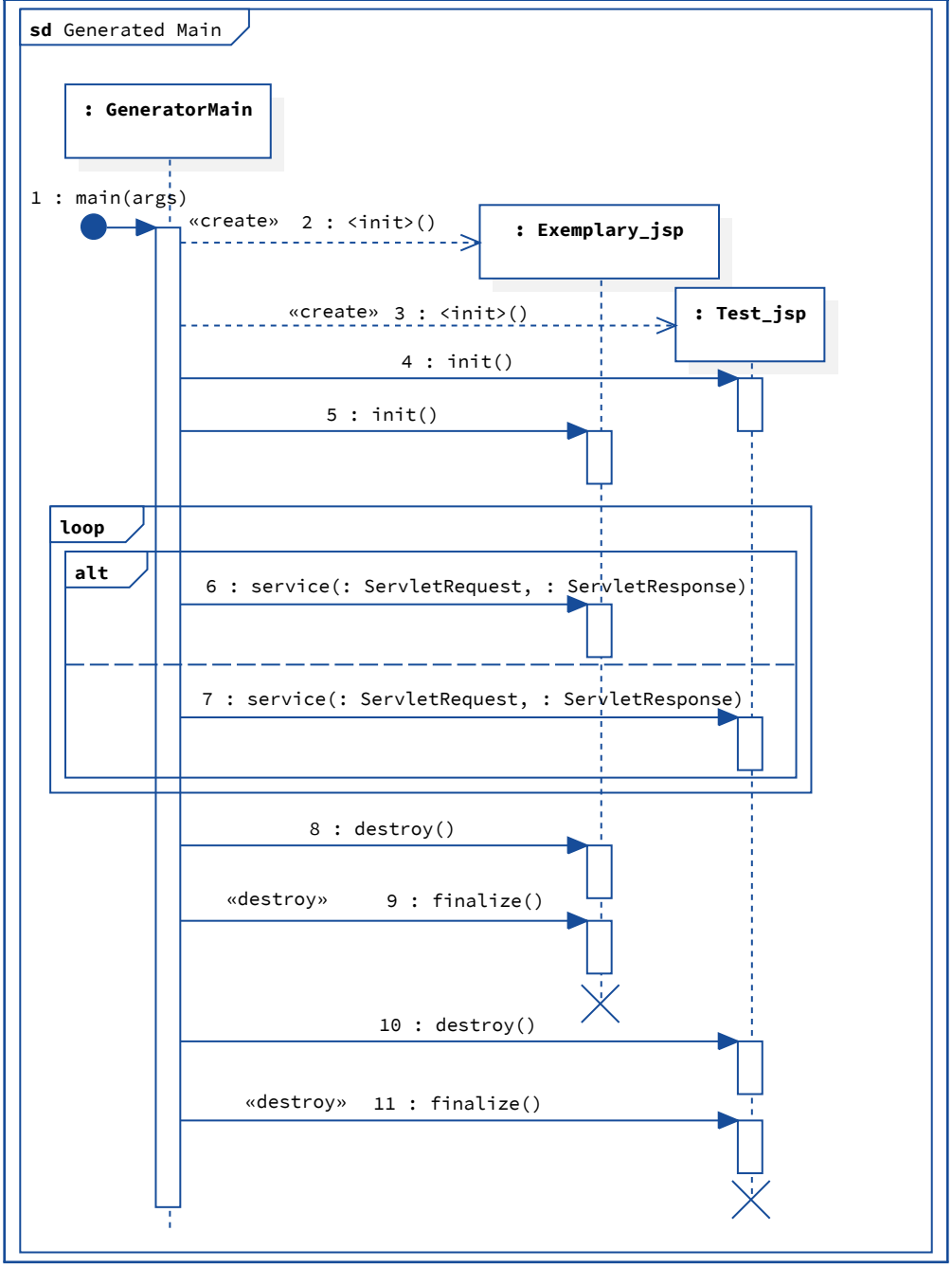
[2]Please note that this behaviour is problematic from a security point of view since an attacker can use this to inject arbitrary HTML and JavaScript code.

Listing 6.2. The generated Java file for the exemplary JSP file shown in Listing 6.1

```
public final class Exemplary_jsp extends HttpJspBase
                                 implements JspSourceDependent {
 public void jspInit() {
   this.log("jspInit()");
 }

 public void jspDestroy() {
   this.log("jspDestroy()");
 }

 public void _jspService(final HttpServletRequest request,
                         final HttpServletResponse response)
       throws IOException, ServletException {

    final PageContext pageContext;
    javax.servlet.jsp.JspWriter out = null;

    try {
      response.setContentType("text/html");
      pageContext = _jspxFactory.getPageContext(this, request, response,
                        null, true, 8192, true);

      out = pageContext.getOut();

      out.write("\n\n<html>\n  <body>\n    <h1>Hello ");
      out.print( request.getParameter("name"); );
      out.write("</h1>\n  </body>\n</html>");
    } catch (java.lang.Throwable t) {
        // error handling omitted
    } finally {
      _jspxFactory.releasePageContext(pageContext);
    }
  }
}
```

two JSP files. The first one is the `Exemplary_jsp` from the examples above, while the second one is called `Test_jsp`. The main method first creates an instance of `Exemplary_jsp` and calls its constructor. Afterwards, an instance of `Test_jsp` is created. After having created all instances, the generator generates calls to all `init` methods. The order of the calls is chosen randomly since the order at runtime is not specified. Next, a loop is generated that calls the

Figure 6.10. UML sequence diagram sketch of the `GeneratorMain`



different `service` methods. A random `service` method is called in each loop iteration to simulate the unpredictable order of user interactions. Finally, the main method destroys the servlets in random order by calling the `destroy` methods.

A slightly extended version of the entry-point generator is available at GitHub[3]. This released version was used in another context to connect the callgraphs of Java-based web service and a Java-based client application to create a cross-application call graph [107].

6.3.2. Callgraph Construction

For constructing a callgraph, ArchSec uses Soot's default callgraph algorithms. Soot provides three different main callgraph construction algorithms, which have various options. The most imprecise construction algorithm uses a class hierarchy analysis [55]. For more precise callgraphs, Soot offers Spark, the Soot pointer analysis research kit [110]. Sparks calculates a callgraph based on points-to sets. It is possible to vary the field sensitivity, how the points-to sets are calculated, and how they are stored. An extension to Spark is the usage of geometric encoding for efficiently storing the points-to information [170]. The last supported callgraph construction framework is Paddle, which uses BDDs to calculate the points-to information [109]. Formally speaking, this phase derives a callgraph CG from a given program P.

Definition 1 (Program) *A program is defined as tuple* $P = (Func, Stmt, funcOf, Call)$*, where* $Func$ *denotes all functions present in the program,* $Stmt$ *all statements,* $Call \subset Stmt$ *denotes those statements that are calls to functions and finally* $funcOf : Stmt \rightarrow Func$ *connects statements to the function they are included in.*

Listing 6.3 shows an exemplary Java program consisting of several classes where the methods call each other. Considering the above mentioned definition, this example program can formally be given as follows:

$$
\begin{aligned}
Func = \{&\texttt{e1}, \texttt{m1}, \texttt{m2}, \texttt{e2}, \texttt{e3}, \texttt{B.<init>}, \texttt{C.<init>}, \texttt{D.<init>}\} \\
Stmt = \{&\texttt{b = new B()}, \texttt{b.m1(u1)}, \texttt{b.m1(u3)}, \texttt{b.m2(p2)}, \texttt{f = 0}, \texttt{f = p}, \\
&\texttt{c = new C()}, \texttt{c.e2(f)}, \texttt{d = new D()}, \texttt{d.e3(p)}\} \\
Call = \{&\texttt{b = new B()}, \texttt{b.m1(u1)}, \texttt{b.m1(u3)}, \texttt{b.m2(p2)}, \\
&\texttt{c = new C()}, \texttt{c.e2(f)}, \texttt{d = new D()}, \texttt{d.e3(p)}\}
\end{aligned}
$$

[3] https://github.com/uni-bremen-agst/javaee-entry-point

Listing 6.3. Running example for component-based callgraph construction

```
class A {
    void e1(int u1, int p2, int u3) {
        B b = new B();

        b.m1(u1);
        b.m1(u3);
        b.m2(p2);
    }
}

class B {
    int f = 0;

    void m1(int p) {
        f = p;
    }

    void m2(int p) {
        C c = new C();
        c.e2(f);
    }
}

class C {
    void e2(int p) {
        D d = new D();
        d.e3(p);
    }
}

class D {
    void e3(int p) {
        // code omitted
    }
}
```

$$
\begin{aligned}
funcOf = \{&(\texttt{b = new B()}, \texttt{e1}), (\texttt{b.m1(u1)}, \texttt{e1}), (\texttt{b.m1(u3)}, \texttt{e1}),\\
&(\texttt{b.m2(p2)}, \texttt{e1}), (\texttt{f = 0}, \texttt{B.<init>}), (\texttt{f = p}, \texttt{m1}),\\
&(\texttt{c = new C()}, \texttt{m2}), (\texttt{c.e2(f)}, \texttt{m2}),\\
&(\texttt{d = new D()}, \texttt{e2}), (\texttt{d.e3(p)}, \texttt{e2})\}
\end{aligned}
$$

As introduced in Section 4.2.2, context-sensitive callgraphs use contexts to distinguish between calls in different state's of the system. Therefore, a *callsite*, the location where a call takes place, consists of a call and the context in which the call is called. Therefore, $Callsite \subset Context \times Call$.

A callgraph can be seen in two ways: one, as a function, storing all functions under a given context a callsite maps to, i.e., where a call leads to. Second, as a graph, where each callsite is connected to exactly those callsites, whose call stems from a function, the callsite mapped to under the above mentioned function. Both definitions are given here:

Definition 2 (callgraph-function) *The callgraph-function of a given program P is a function, mapping a callsite to all functions it adresses:*

$$call : Context \times Call \supset Callsite \to \mathcal{P}\left(Context \times Func\right) \tag{6.1}$$

Please note, that the co-domain is the power set of $Context \times Func$ as a call can—even under the same context—map to multiple methods, depending on the preciseness of the used callgraph construction algorithm (c.f. Section 4.2.2). Following the *callgraph-function call* for all callsites and listing all calls present in the receiving functions leads to building the actual callgraph as follows:

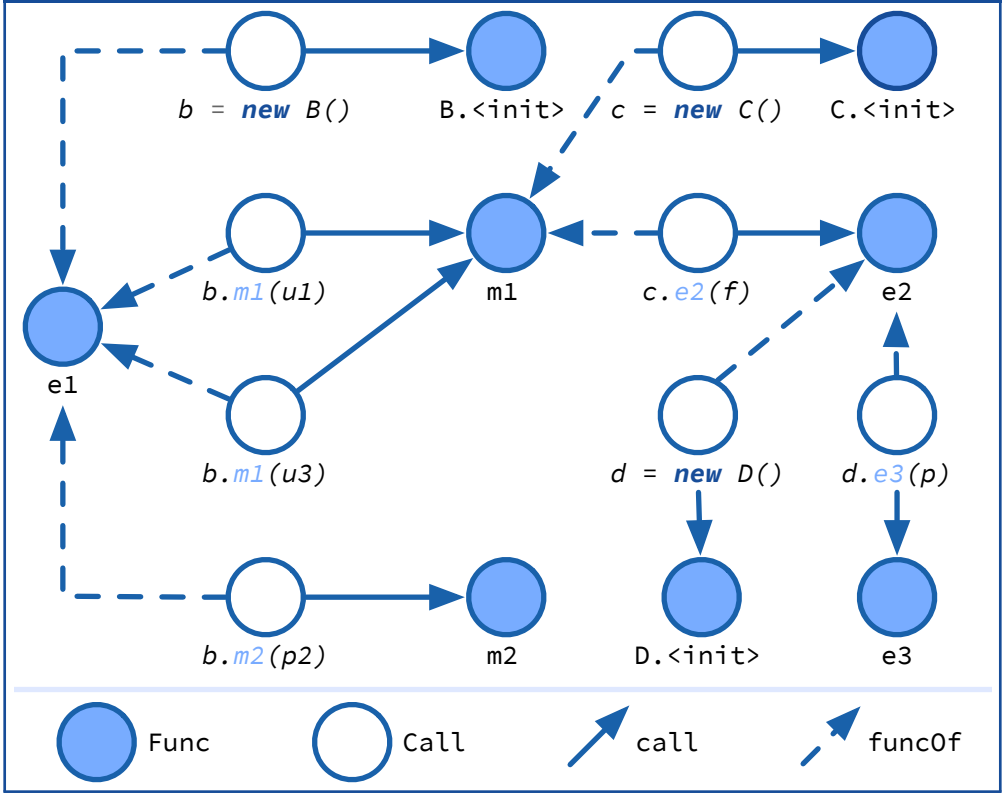
Definition 3 (callgraph) *The callgraph of a given program $P = (Func, Stmt, funcOf, Call)$ and its callgraph-function call is a graph consisting of functions in their context $(ctx, f) \in Context \times Func$ and callsites $cs \in Callsite$ connected by edges representing the mappings $funcOf$ and call:*

$$\begin{aligned} CG &= \left(Callsite \cup \left(Context \times Func\right), E\right) \\ E &= \{(cs, (ctx, f)) | cs \in Callsite, (ctx, f) \in call(cs)\} \\ &\cup \left\{((ctx', f'), cs') | funcOf(cs'_{|2}) = f'\right\} \end{aligned}$$

Please note that $x_{|i}$ denotes the ith component of a tuple x.

Figure 6.11 shows the callgraph resulting from Listing 6.3. Please note that the depicted callgraph ignores context for the sake of clarity. Furthermore, statements that are not calls are left out of the callgraph to concentrate on the results of the callgraph function. Another valid defintion might include all statements (instead of calls) in their respective contexts.

Figure 6.11. Callgraph visualisation of the running example shown in Listing 6.3



6.3.3. Architecture Recovery

The architecture recovery process extracts information on components, their entries, and their connection. The process splits into four steps:

Component detection The framework-specific architecture recovery component detection subdivides the analysed software into components based on the framework they use. For this task, the framework's documentation provides information on the way of declaring components. Usually, this can be reached by adding deployment information, implementing specific interfaces, or annotating classes. Additionally, the process employs information on the technical domain's specific architectural pattern during architecture reconstruction. All components C make up the set $\mathcal{C}$. For Listing 6.3, the component detection obtains $\mathcal{C} = \{C_1, C_2, C_3\}$.

Component entry detection The recovery process identifies entry points for the components found. A component entry point can be, as long as the framework documentation does not state otherwise, any public method within the component's classes. Java frameworks usually use reflection to call these entr points. The application entry points identified in the callgraph preparation phase are a subset of the identified component's entry points. The application's entry points are explicitly marked since they are user controlable and accessible from outside the application, and an attacker can therefore call them. This leads to the denomination of $Entry_C\ \forall C \in \mathcal{C}$. Obviously, it holds that $Entry_C \subset Func_C \subset Func$ when $Func_C$ denotes all functions in component C. Considering Listing 6.3, a configuration might yield $Entry_{C_1} = \{\texttt{e1}\}, Entry_{C_2} = \{\texttt{e2}\}, Entry_{C_3} = \{\texttt{e3}\}$.

Callgraph mapping A component's entry points are starting points for the following callgraph mapping. This step traverses the callgraph, starting at the entry points and stopping at other components' entry points. The mapping assigns all visited methods to the component of the entry point's component. This step thus identifies which part of the callgraph belongs to which component. Please note that the mapping may map a method to multiple components, for instance, methods belonging to Java's JDK. This component-based callgraph mapping requires a refinement of Definitions 2 and 3, restricting the callgraph function as well as the callgraph to the given component.

Definition 4 (component-based callgraph-function) *The callgraph-function of a given program P restricted to a component C is defined as*

$$call_{|C} : Callsite_{|C} \rightarrow \mathcal{P}(Context \times Func) \tag{6.2}$$

where $Callsite_{|C} = \{cs \in Callsite | funcOf(cs_{|2}) \in Func_C\}$.

The same restriction holds for the callgraph leading to the following

Definition 5 (component-based callgraph) *A component-based callgraph for a component C is defined as:*

$$\begin{aligned} CG_C &= \{Callsite_{|C} \cup (Context \times Func_C), E_C\} \\ E_C &= \{e = (cs, (ctx, f)) \in E | cs \in Callsite_{|C}, f \in Func_C\} \\ &\quad \cup \{e' = ((ctx', f'), cs') \in E | f' \in Func_C, cs' \in Callsite_{|C}\} \end{aligned}$$

There are two issues with these definitions worth discussing: First, while this definition precisely states a component-based callgraph, $Func_C$ is unknown. Second, the definition of a component-based callgraph does not contain the exit calls of the component, as their receiving node is no longer in $Func_C$ and

therefore, the edge is neither (otherwise, the callgraph were no graph). The definition using the component-based callgraph-function, however, still yields component-leaving edges, as one can still call $call_C$ on the *last* calls in $Callsite_C$. As the outgoing edges are necessary for the construction of an EDFD later on, the computation of $Func_C$ will be done on the basis of the *callgraph-function*.

To compute $Func_C$, it is necessary to define the equivalent of a graph traversal - the concatenation of *call*. However, as has been seen above, *call* has different sets for domain and co-domain, therefore, the canonical approach will not suffice. Nevertheless, using $funcOf$ it is possible denote $call^n$ as following the callgraph function n times.

Definition 6 *Let $(ctx, c) \in Callsite$ be an input to the function call.*

$$\begin{aligned}(call^2)(ctx, c) &:= call \circ (funcOf \circ call)\,(ctx, c)\\ &= call(funcOf(call(ctx, c))\\ &= call(\{(ctx', c') \in Callsites | (ctx', funcOf(c')) \in call(ctx, c)\}\end{aligned}$$

Calling the callgraph function n times is then a trivial extenstion: $call^n(ctx, c) = (call \circ (funcOf \circ call)^{n-1})(ctx, c)$. Please note the exponent $n-1$, as the nth call of the function call is already ensured through the leftmost call of the function call. This notation ensures that the co-domain of $call^n$ equals the one of call.

Please assume the standard notation of $g(A) = \bigcup_{a \in A} g(a)$ to denote the use of a function on a subset of its domain instead of an element. Now, $Func_C$ can be computed as the set of all functions that are called starting from the entry points of a component and are not themselves entry points to another component.

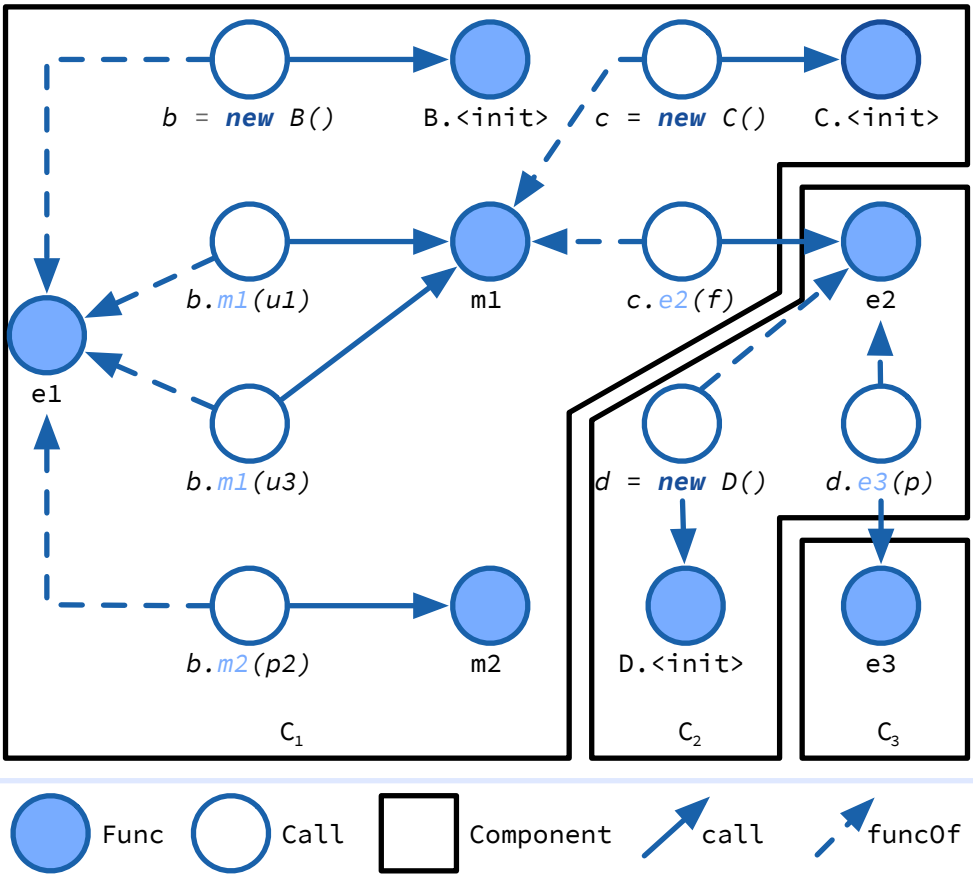
Theorem 1 *Let $C^* \in \mathcal{C}$ be an component of the given program P. Then $Func_{C*}$ can be retrieved as:*

$$\begin{aligned}Func_{C^*} = Entry_{C^*} \cup \{f \in Func |\\ &\exists e \in Entry_{C^*} : funcOf^{-1}(e) \cap Call = Call_e \neq \emptyset\\ &\wedge \exists n \in \mathbb{N}, \exists c \in Call_e, \exists ctx, ctx' \in Context :\\ &(ctx', f) \in call^n(ctx, c)\\ &\wedge \forall C \in \mathcal{C}, C \neq C^* : f \notin E_{C^*}\}\end{aligned}$$

The first condition in the above conjunction is necessary to extract all calls from the entry points to make sure, functions are called from an entry point of this

component. The second condition follows the callgraph-function *call* through the callgraph (every function along the way is element of $Func_C$) and the third condition ensures that the traversal of the callgraph stops (excluding) when the call-receiving function is an entry function to another component.

Figure 6.12. Component callgraph visualisation of the running example shown in Listing 6.3



Applying Theorem 1 to the example from Listing 6.3, $Func_{C_1}, Func_{C_2}, Func_{C_3}$ can be computed as follows:

$$Func_{C_1} = \{\texttt{e1}, \texttt{B.<init>}, \texttt{m1}, \texttt{m2}, \texttt{C.<init>}\}$$
$$Func_{C_2} = \{\texttt{e2}, \texttt{D.<init>}\}$$
$$Func_{C_3} = \{\texttt{e3}\}$$

To illustrate the above computation, Figure 6.12 shows the component-based callgraphs (or rather, the separation of the original callgraph into three component-based callgraphs) and thus the resulting *Func*-sets. The component-based callgraphs stop at the call that would direct to another component's entry point.

External communication detection While the previous step identifies inter-component communications, it is still necessary to identify communications with external systems or file access. For this purpose, different pattern-based detection algorithms exist, supporting different Java APIs and the framework APIs. Two examples, working on all platforms, are the *opened file identification* and the *URL connection identification*. Both analyses search for specific method calls and then try to determine actual parameter values to identify the opened file or the accessed server using interprocedural backward dataflow analyses. Finally, they map the information to the components depending on the location of the initially detected call and the identified dataflows. As already mentioned, a method may be mapped to multiple components. Therefore, the analysis information may be assigned to multiple components as well.

The *URL connection identification*, for instance, searches for calls to the `openConnection` method of the `URL` class. Next, the analysis determines the actual name of the opened URL using an intraprocedural backward analysis. The actual server address might not be determined for different reasons, for instance, the actual value might be dynamic or stem from some configuration file. In such a case, to not loose possible communication paths, the analysis adds a connection to a server that is labelled as unknown. Consequently, such communication paths need a proper manual investigation.

6.3.4. Security Fact Extraction

The security fact extraction consists of different independent analyses, which search for the use of security measures. The analyses focus on framework-specific or JDK-specific APIs and their usage. In particular, the currently implemented analyses are:

TLS/SSL validity The TLS/SSL check [36] is a pattern-based analysis. The analysis searches for classes implementing Java's `TrustManager` and `HostNameVerifier`, which are known to have the potential of breaking TLS/SSL security [68]. The analysis checks for different patterns that result in a broken implementation, such as empty bodies, implementations that do not throw the required `CertificateException`, or implementations that only return specific

values. Afterwards, the analysis searches the program code for usage of these implementations. According to the findings, the TLS-secured channels of the using component are marked as insecure.

The `TrustManager`'s purpose is to validate whether the presented public key certificate is trustworthy. Therefore, a trust manager must check if the certificate is not expired, revoked, and signed by a trustworthy root authority. Failing to check these properties properly may result in an insecure connection that an attacker might hijack. Different internet sources, such as StackOverflow[4], suggested using broken trust managers to solve problems with self-signed certificates.

Listing 6.4. TrustManager implementation accepting all presented certificates

```
import javax.net.ssl.X509TrustManager;

private class AllowAll implements X509TrustManager {
    @Override
    public void checkClientTrusted(final X509Certificate[] chain, final
    ↪  String authType) throws CertificateException {
    }

    @Override
    public void checkServerTrusted(final X509Certificate[] chain, final
    ↪  String authType) throws CertificateException {
    }

    @Override
    public X509Certificate[] getAcceptedIssuers() {
        return new X509Certificate[0];
    }
}
```

Listing 6.4 shows a very simple broken trust manager. The implemented method `checkServerTrusted` does not conduct any checks and therefore does not throw any `CertificateException`. Consequently, an attacker conducting a man-in-the-middle attack can present any certificate, and an application using the trust manager implementation will accept it.

The `HostnameVerifier`'s task is to check whether the hostname the application is communicating with matches the hostname given in the certificate. This step

[4] https://stackoverflow.com

is necessary to prevent attackers from using an officially signed certificate for a different domain. There are several strategies for accepting hostnames, for instance, if a certificate for a domain is accepted for sub-domains as well. A hostname verifier has to implement a certain method and return a boolean value if the hostname matches the certificate or not.

Listing 6.5. `HostnameVerifier` implementation accepting all host names

```
import javax.net.ssl.HostnameVerifier;
import javax.net.ssl.SSLSession;

public class CustomVerifier implements HostnameVerifier {
    public boolean verify(final String hostname, final SSLSession session)
    ↪ {
        return true;
    }
}
```

Listing 6.5 presents a simple `HostnameVerifier` implementation that ignores the certificate settings and simply returns **`true`**. Therefore, a such created connection might be hijacked by an attacker.

The static analyses classify implementations of the relevant interfaces. They check if the implementation throws the required exception or returns **`false`**. If they do not, they are classified as insecure. Furthermore, they check if the relevant parameters are used, such as the hostname or the certificate chain. Additionally, the analysis knows the categories of the default implementations of these interfaces. Afterwards, the analyses search for uses of these implementations and use the categorisation to rate the security of an identified connection. If an implementation does not match one of the known behaviours it is categorised as *review required*.

Encryption usage The encryption analysis [175] uses two inter-procedural Vasco-based analyses phases. Java provides with the Java Cryptographic API a unified API for all encryption algorithms. The `Cipher.getInstance` method works as a factory function to create `Cipher` objects. The parameter of this factory method determines the used cryptographic algorithm, while a subsequent call of the `init` method configures the algorithm. The first phase of the analysis identifies calls to the factory method and tracks the returned `Cipher` object using an inter-procedural dataflow analysis. The trace is then used to find all interactions with the object, creating an object-process graph for the `Cipher` object. In the second phase, an inter-procedural backward analysis is used

to identify the parameter values of the factory call and the initialisation call. This analysis step allows to reconstruct the actual configuration of the `Cipher` object. Based on the extracted information, the security of the used encryption algorithm is rated. Based on the extracted information, the security of the used encryption algorithm is rated using BSI TR-02102 [76]. The technical guideline specifies which parameter sets, e.g. mode, padding, and key size, are secure for specific encryption algorithms. Furthermore, the rating takes the cryptographic key generation into account. Several programs have hardcoded keys, making it easier for an attacker to get the key if he has access to the program's binary.

Listing 6.6. Exemplary code snippet of an analysed Android app

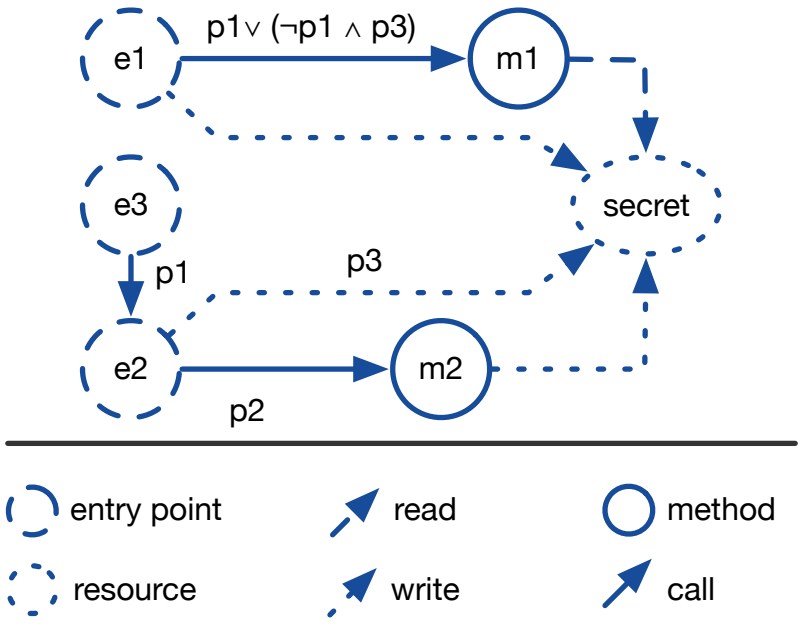
```
import java.security.cert.CertificateFactory;
import javax.crypto.Cipher;
import javax.security.cert.X509Certificate;

public class CryptoExample {
  private X509Certificate certificate;

  public void create() {
    InputStream is = getClass().getResourceAsStream("/assets/test.pem");
    CertificateFactory fact = CertificateFactory.getInstance("X.509");

    this.certificate = (X509Certificate) fact.generateCertificate(is);
  }

  public byte[] encrypt(byte[] data) throws Exception {
    Cipher cipher = Cipher.getInstance("RSA/ECB/PKCS1Padding");
    cipher.init(Cipher.ENCRYPT_MODE , certificate.getPublicKey());

    return cipher.doFinal(data);
  }
}
```

Listing 6.6 shows an exemplary encryption usage identified by the analysis in an Android app. The `create` method loads a cryptographic key from the application's binary. Afterwards, the `encrypt` method uses the RSA algorithm in the electronic codebook mode with PKCS1 padding to encrypt data. The analysis categorises the algorithm configuration as secure. Nevertheless, the used key material is problematic, which is correctly identified by the analysis. It is possible to configure the categorisation of API methods. The method `getResourceAsStream()` is configured to return a constant value (the loaded certificate is part of the distributed binary). Accordingly, the analysis identifies

that the used key material is a constant value.

Authorisation The authorisation analysis [34] identifies how calls and attribute accesses are protected using authorisation means. First, it identifies calls to policy decision points and tracks the authorisation decision using inter-procedural dataflow analysis to identify all variables that hold authorisation decisions. For each variable, traces are calculated containing the flow of the authorisation decisions. In the second step, the `if`-conditions are identified which use authorisation decisions. Third, the parameters of the policy-decision point calls are calculated using an inter-procedural backward analysis. For the parameter values, the analysis creates flows similar to the ones calculated in the first step. The traces of the first and the last step are then matched to identify the authorisation fact enforced by the `if`-conditions. In the last step, the statements guarded by the `if` statements are calculated using an intra-procedural analysis. Based on these results, the analysis calculates the authorisation facts, e.g. permission, that must hold to call methods or get access to assets.

Figure 6.13. Extracted access graph



Listing 6.7 shows an exemplary code snippet that uses Apache Shiro[5] for its authorisation. The policy decision point in this example is Apache Shiro's

[5] https://shiro.apache.org

`isPermitted` method. It checks if the subject has the permission label that is passed as its first parameter. Figure 6.13 shows the extracted access graph of the analysis that is mapped to the component callgraphs described earlier.

6.3.5. Summary

This section elaborated on the extraction of system models. It first describes the problem of missing main methods that occur when analysing component-based software systems and the solution ArchSec uses to tackle this problem. Furthermore, it describes the architecture recovery process for component-based software systems, the extraction of security aspects, and how they are matched to each other.

According to the categorisation of Ducasse and Pollet introduced in the Prelude, the goal of the described architecture recovery is redocumentation and analysis. The utilised inputs are source code and additional textual software artefacts. The employed detection process is a hybrid approach since it refines the architecture pattern, introduces architectural components, and merges source code artefacts to the identified components. Furthermore, it is a semi-automatic, abstraction-based technique that maps low-level information to high-level abstractions using a programmed abstraction employing graph matching. The resulting output is intended for visual software views, the visualisation of the extracted extended dataflow diagrams, and conformance checking. The planned identification of security flaws and their mitigations are, essentially, a conformance checking.

6.4. Chapter Summary

This chapter raised the requirements for the static analysis, introduced ArchSec's system model, and the static extraction of a software's architecture and security information. Concerning the research questions, this chapter deals with three of them. The section on system model extraction contributed to *"RQ3—What kind of architectural information can be automatically extracted for existing software systems?"* and explained how ArchSec extracts a software's architecture by identifying components, entry points, and exits. Furthermore, this chapter contributes to *"RQ4—Which architectural security information can be automatically extracted using static software analysis?"* and explains how information on different security aspects, such as TLS, encryption, and access control, are extracted.

Listing 6.7. Authorisation example using Apache Shiro

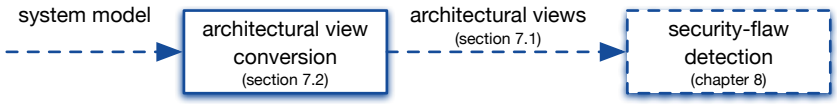
```
import org.apache.shiro.SecurityUtils;

class Authorisation {
    int secret = 0;

    void e1() {
        if(check("p1") || check("p3")) {
            m1();
        }

        secret = 12;
    }

    void e2() {
        if(hasP2()) {
            m2();
        }


        if(check("p3")) {
            secret = 12;
        }
    }

    void e3() {
        if(check("p1")) {
            e2();
        }
    }

    boolean hasP2() {
        return check("p2");
    }

    void m1() {
        System.out.println(secret);
    }

    void m2() {
        secret = 27;
    }

    boolean check(String p) {
        return SecurityUtils.getSubject()
            .isPermitted(p);
    }
}
```

Chapter 7

Architectural Views

The last chapter explained ArchSec's system model's static extraction, comprising architectural information, potential protection goals, and security measures. This chapter follows ArchSec's workflow and sheds light on the next step, the architectural view creation, as shown in Figure 7.1. Therefore, Section 7.1 discusses *"RQ1—How can dataflow diagrams be enhanced to support a more formal expressiveness?"* and introduces extended dataflow diagrams, ArchSec's representation of security-relevant architectural views and the basis of the automatic security flaw detection. Next, Section 7.2 explains the system model's conversion into extended dataflow diagrams using model-to-model transformation techniques. Finally, Section 7.3 introduces formal extended dataflow diagrams. The thesis author's academic publications [36–39, 114] already describe different aspects of extended dataflow diagrams. The following chapter will explain extended dataflow diagrams in full details.

Figure 7.1. Outline of the architectural view conversion



7.1. Extended Dataflow Diagrams

This section introduces extended dataflow diagrams—an extension to traditional dataflow diagrams that can store information on security requirements and security measures. First, subsection 7.1.1 illustrates the data model of extended dataflow diagrams and explains the model's ideas. Afterwards, Subsection 7.1.2 presents the standard extended dataflow diagram schema that helps to describe software systems.

7.1.1. Data Model

This subsection explains the EDFD's data model used in ArchSec. The design goals of this model are a higher formality of dataflow diagrams and the capability of automated threat modelling. Therefore, ArchSec's data model is divided into two conceptual levels. Figure 7.2 and Figure 7.4 depict the two different levels of the data model. While the original dataflow diagrams (cf. Section 3.3) use a fixed number of different node types, such as processes, data stores, and external entities, extended dataflow diagrams use an extensible schema, allowing the definition of new node types and channel types. Furthermore, they introduce attributes for describing protection goals and security measures.

The first level, called *EDFD schema*, corresponds to a meta-model in model-driven software development. This level determines the expressiveness of an EDFD by allowing the introduction of concrete attribute types and diagram element types. In this level, a `Diagram` contains exactly one `Schema`. The schema itself contains `ChannelAttributeTypes`, `DataAttributeTypes`, `ElementAttributeTypes`, and `TrustAreaAttributeTypes`. Each of these classes is a specialised `AttributeType`, having a `name` and a `kind`. An attribute type can be refined by other attribute types, allowing a hierarchical order. Furthermore, a schema contains `ChannelTypes`, `DataTypes`, `ElementTypes` and `TrustAreaTypes`, which are specialised `Types`. A `Type`, in turn, has a name and is `refined by` other types with the same generic parameter type binding. Additionally, a type implies `AttributeBindings`. An attribute binding maps an attribute type to a `value`. The value type depends on the attribute's type `kind`.

To summarise, the EDFD schema allows defining hierarchical attribute types expressing both protection goals and security measures. Besides, it allows defining types for visual diagram elements to customise diagrams. A type can imply attribute bindings that bind an attribute type to a concrete value for all diagram elements of this type. Figure 7.3 shows a UML object diagram of an exemplary EDFD to illustrate the model's semantics. The object diagram models a single data type named *personal information*. There are two element types in the model. A more concrete element type *server* refines the generic element type *device*. Finally, the three different channel types are from most generic to most specific: *inter-process communication*, *HTTP*, and *HTTPS*. Furthermore, the object diagram contains two attribute types. First, the `boolean` data attribute type named "*is confidential*" and, second, the `boolean` channel attribute type "*is encrypted*". *Personal information* implies the attribute binding from "*is confidential*" to the value `true`. While *inter-process communication*, in general,

Figure 7.2. UML class diagram of the EDFD

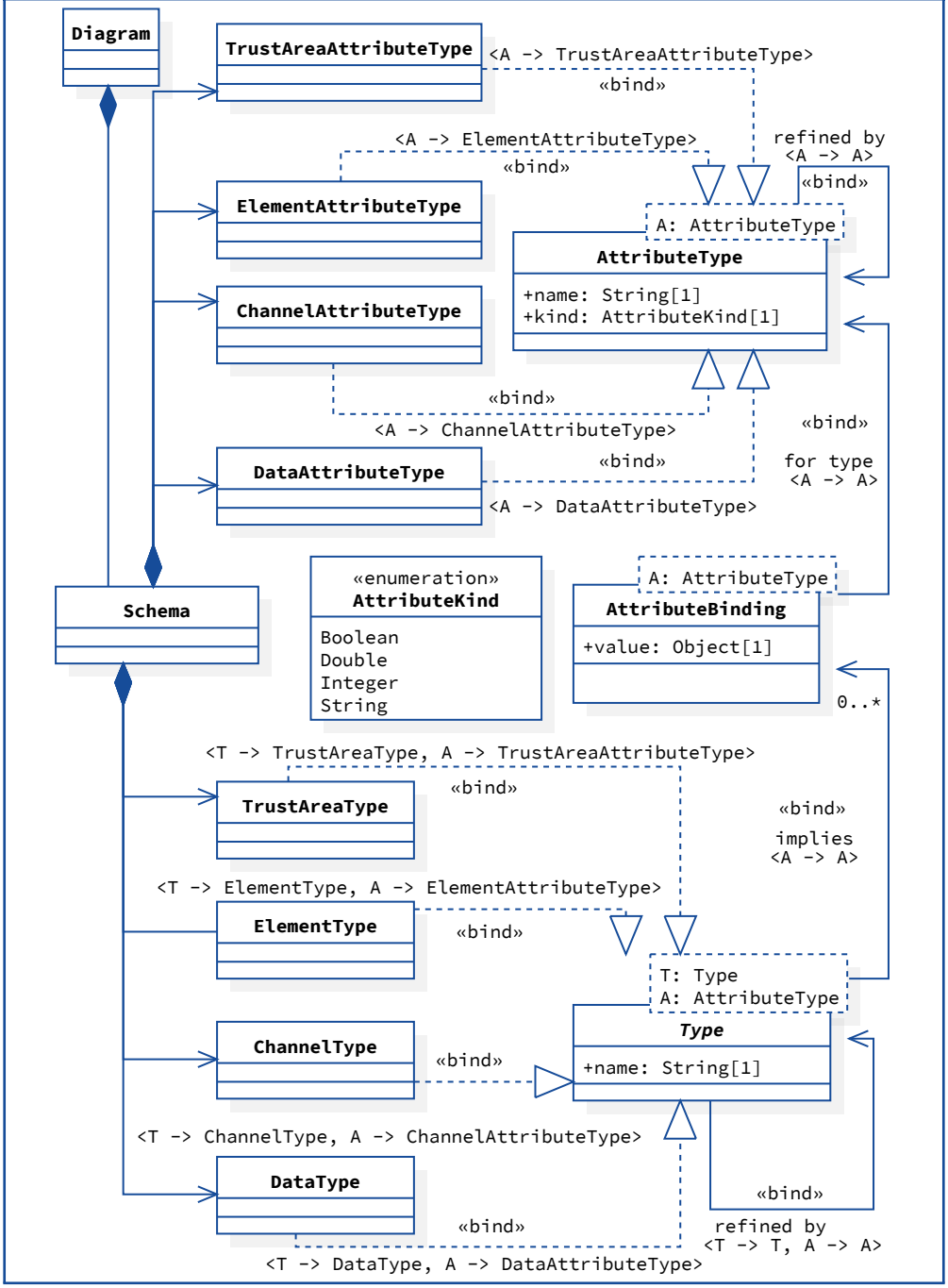
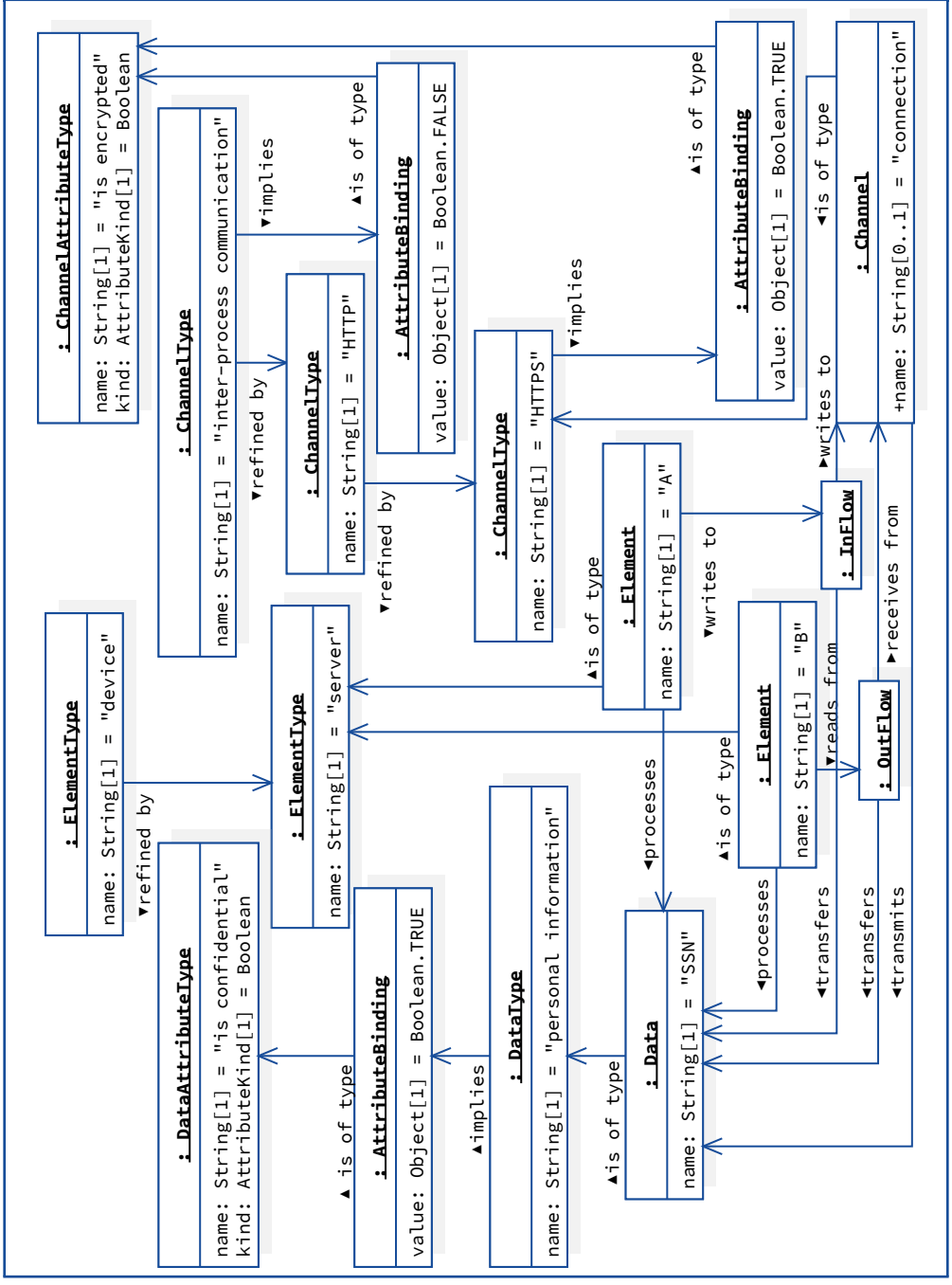


Figure 7.3. UML object diagram of an exemplary EDFD



implies the attribute binding from *is encrypted* to the value `false`, the *HTTPS* channel type overwrites the implication. It binds the channel attribute type to the value of `true`.

Figure 7.4 depicts the remaining part of extended dataflow diagrams. The second level corresponds to a model in model-driven software development and allows one to specify a concrete EDFD. A `Diagram` contains an arbitrary number of `AttributeHost`s, which contain the already introduced `AttributeBinding`s. `AttributeHost`s are `Channel`s, `Data`, `Element`s and `TrustArea`s. The data element models any information existing in the software system. It has a `name`, can be decomposed hierarchically, and is of a specific data type. Next, `TrustArea`s allow specifying different levels of trust in an EDFD. They have a `name` and can be refined hierarchically. Trust areas have a type as well and may `contain` an arbitrary number of `Element`s. Like the modelling elements above, elements can be decomposed hierarchically and have a corresponding element type. Additionally, an element `processes` data, `sends` data to `InFlow`s, and `receives` data from `OutFlow`s. `InFlow`s and `OutFlow`s might be named and link the data the element `sends` or `receives`. Finally, `Channel`s represent communication channels between elements. A channel might be named and has an associated channel type. It `transmits` the data that the `InFlow`s `write to` it and the `OutFlow`s `receive from` it.

Figure 7.5 shows the exemplary EDFD defined in Figure 7.3. The object diagram contains objects of second-level elements as well and specifies concrete *personal information* named *SSN*, the abbreviation for *Social Security Number*. Furthermore, it contains two elements of type *server*, named *A* and *B*. Server *A* writes the *SSN* to an *HTTPS*-typed channel, and *B* receives the *SSN* from the channel. In essence, this models two server elements where one of the servers sends an *SSN* to the other.

7.1.2. Extended Dataflow Diagram Schema

Whilst the last section introduced the EDFD model, this section delves into the role and influence of an EDFD's schema since it plays a crucial role in the expressiveness and semantics of the diagram. This subsection introduces a domain-specific language to describe EDFD schemas within ArchSec. Since the described schema concept makes EDFDs very adaptable, this subsection sheds light on how ArchSec's predefined schema looks like. However, before examining ArchSec's built-in schema, this subsection shows a schema allowing one to express DeMarco's traditional dataflow diagrams (please compare Section 3.3). Finally, this subsection summarises the information on extended dataflow diagram schemas.

Figure 7.4. UML class diagram of the EDFD cont'd

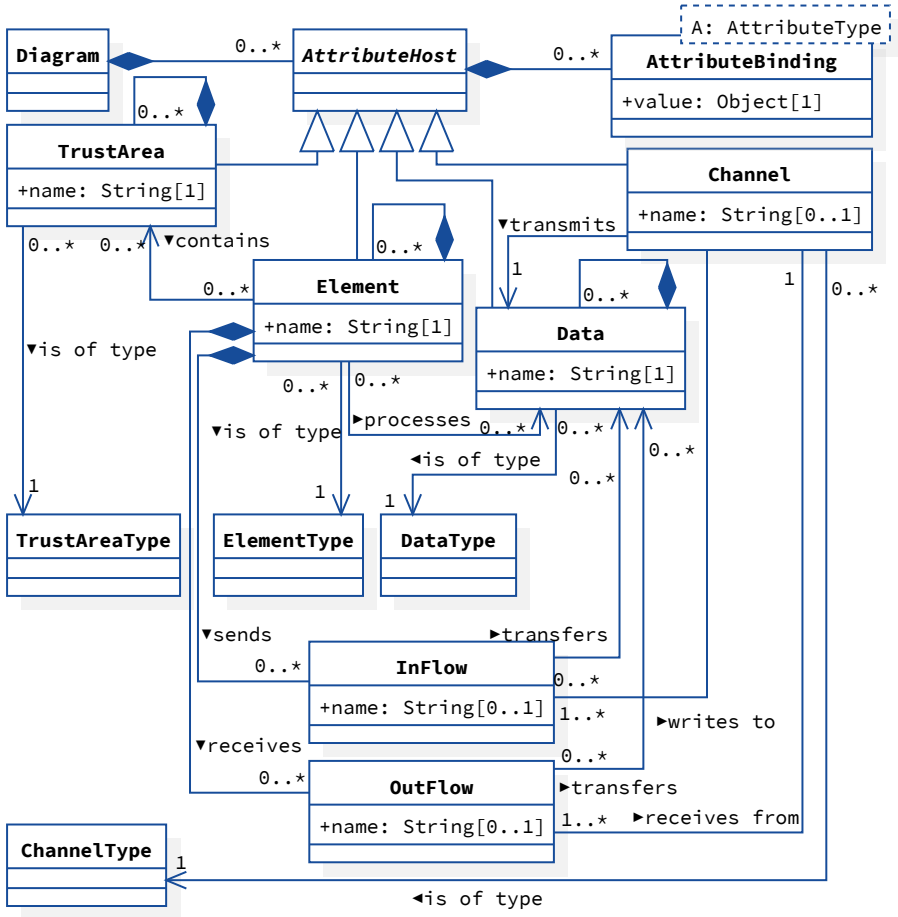
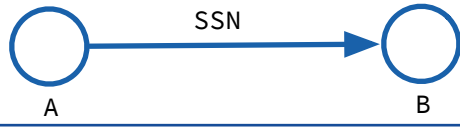


Figure 7.5. Extended dataflow diagram sketch of the object diagram shown in Figure 7.3



Domain-specific language for schemas

In model-driven software development, domain-specific languages give the possibility to capture the unique intricacies of particular problems. In contrast

to general-purpose languages, they regularly do not allow solving all kinds of computational problems. However, they help to express their problem more abstractly than in a general-purpose language. Since ArchSec uses different schemas based on the analysed application, the extracted security aspects, and other aspects, it supports a domain-specific language for quickly defining and extending schemas.

Listing 7.1. Schema definition for the types introduced in Figure 7.3

```
schema {
  types {
    ChannelType "inter-process communication" {
      children {
        ChannelType "HTTP" {
          children {
            ChannelType "HTTPS" {
              implies { "is encrypted" : true }
            };
          }
        };
      }
      implies {"is encrypted" : false}
    };

    DataType "personal information" {
      implies { "is confidential" : true }
    };

    ElementType "device" {
      children {
        ElementType "server";
      }
    };

    ElementType "process";
  }

  attributes {
    ChannelAttribute "is encrypted" of type Boolean;
    DataAttribute "is confidential" of type Boolean;
  }
}
```

The domain-specific language designed for schema specification does not allow the definition of elements from the concrete model level, depicted in Figure 7.4.

Instead, it is possible to specify instances for all modelling elements shown in Figure 7.2. Listings 7.1 shows a schema written in ArchSec's schema definition language. The shown schema contains all types shown in the object diagram in Figure 7.3. As the listing shows, the language is straightforward since it simply lists all existing types and specifies existing attribute implications. ArchSec's primary design goal is flexibility and extensibility. Thus, it is possible to split the schema into several files, allowing a limited core schema and the extension of this core schema with framework- or language-specific schema elements, such as Android-specific parts, in separate files.

A traditional dataflow diagram schema

The schema mechanism allows customising extended dataflow diagrams to be fully compatible with traditional dataflow diagrams, such as the ones used by Snyder and Swiderski. Listing 7.2 shows a very simple schema defining the elements used by them. The schema defines a single channel type and a single trust area type. Furthermore, it defines an element type for each node-like element in Swiderski's and Snyder's dataflow diagrams, namely *data store*, *external entity*, *multiple process*, and *process*. While it is possible to use extended dataflow diagrams in this way, it is unrecommended since it neglects all advantages extended dataflow diagrams offer.

Listing 7.2. Schema for supporting Swiderski's and Snyder's dataflow diagrams

```
schema {
  types {
    ChannelType "dataflow";

    ElementType "data store";
    ElementType "external entity";
    ElementType "multiple process";
    ElementType "process";

    TrustAreaType "boundaries";
  }
}
```

ArchSec's built-in schema

ArchSec offers built-in schemas that build on top of each other and that were developed during different research projects. The most fundamental schema defines attributes according to the protection goals introduced in Section 2.1. Appendix A shows the schema written in the presented schema definition language. Besides the protection goals, it defines attributes for specifying countermeasures, such as channel encryption, introduced in the information security section. Furthermore, it introduces some basic types of channels, data, elements, and trust areas, allowing to model software systems rudimentarily. ArchSec contains schema extensions adding specific resource types, such as different file types or databases. It supports schema extensions for modelling static software structures, such as the call graph or dataflows within software systems. There even is a schema extension allowing to model BPMN diagrams introduced in Section 3.3. To ensure a more fluent reading experience, an explanation of these extensions follow where they are put to use.

Schema summary

This section explained the concepts of the extended dataflow diagram schema and showed how ArchSec notes schemas. Furthermore, it showed that the schema makes extended dataflow diagrams flexible and customisable, allowing one to express different aspects of software systems, such as traditional dataflow diagrams or BPMN diagrams. Adding attributes to types eases the use of extended dataflow diagrams for non-security experts since the built-in schema contains much security knowledge. Using predefined types, such as the data type *credentials* or the channel type *HTTP*, enables the automatic security flaw detection without the user knowing about security details.

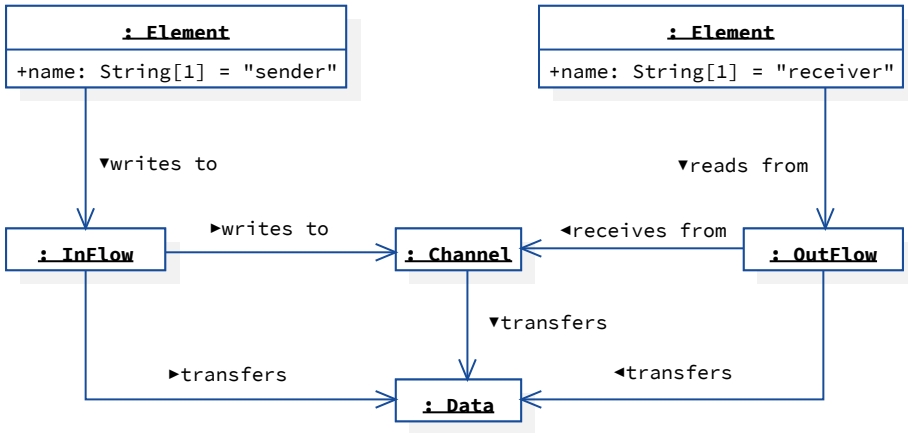
7.1.3. Since (not) all Channels are Created Equal

In computer science, there are different ways of communication. Applications can choose to use asynchronous communication whether there is no need for bidirectional communication. If such communication is built on top of the User Datagram Protocol (UDP), the sending application does not even know whether the message was transmitted successfully. A synchronous communication, on the other hand, allows two applications to exchange information. Depending on the protocol design decisions, both applications can send information when necessary, or this is restricted to one application. The other application can then only reply to sent requests. The most commonly known protocol using such a communication schema is HTTP, where clients establish a connection to a server

and send their requests, while the server only sends responses. In some situations, communication is not only bilateral. However, in some situations, there is the requirement to send information to multiple other applications, broadcast or multicast communication can be used. These types of communications send a message to multiple machines, either using some subnet masks or using their exact IP addresses. Lastly, there are middleware systems allowing n sending applications to send messages to m receiving applications.

There are different possibilities to model all these communication methods in extended dataflow diagrams coming with different advantages and disadvantages. First, it would be possible to introduce different channels, one for each communication method, resulting in a larger number of modelling elements. Second, it would be possible to use a channel to model a simple uni-directional dataflow. However, the downside of this approach is the explosion of channels since simple request and response communication results in two channels, and an *m–to–n* communication results in, at least, $m \cdot n$ channels for asynchronous messages and $2 \cdot m \cdot n$ channels for request-and-response-like communication. To counter these disadvantages, ArchSec chose a third alternative.

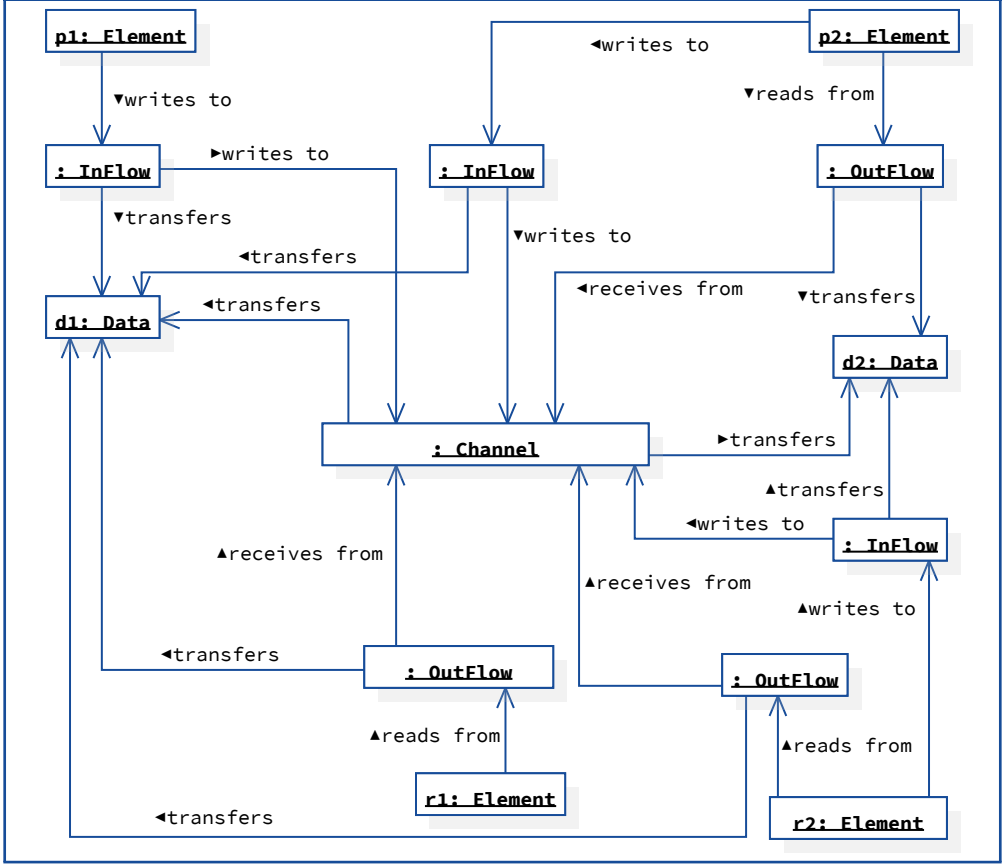
Figure 7.6. EDFD excerpt for an unidirectional channel



ArchSec's extended dataflow diagrams have a single channel concept, but each channel can have, as described earlier, several `InFlows` and `OutFlows`. The chosen channel concept allows modelling all the presented communication mechanisms with the same modelling concept. Figure 7.6 shows an excerpt of an extended dataflow diagram, where a sender sends some data to a receiver and no reply is sent. As the receiver has no connected `InFlow`, he is not able to write data to the channel. Figure 7.5 already showed the modelling of bidirectional

communication between a server and a client.

Figure 7.7. EDFD excerpt for an multidirectional channel

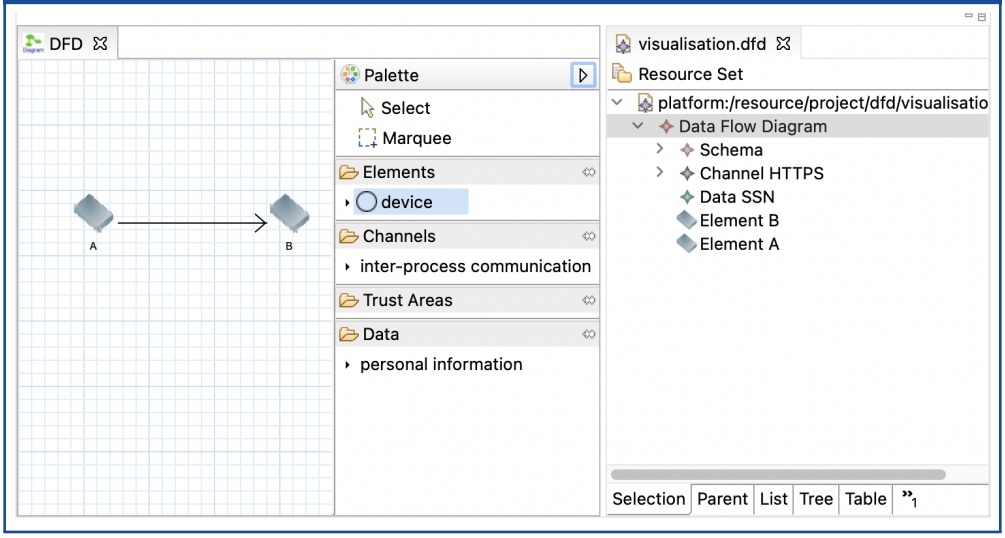


Finally, Figure 7.7 shows a more complicated channel that can be realised using a communication middleware, such as the Java Message Service (JMS) [54]. JMS offers a publish and subscribe model, where producers publish messages under a topic. The middleware then delivers the message to all receivers subscribed to this topic. The example consists of two producers, named *p1* and *p2*, and two receivers, named *r1* and *r2*. The producers write the data *d1* to the channel. The two receivers read the data from the channel, and the receiver *r2* sends a reply message named *d2* for the message received from *p2*. Accordingly, the middleware delivers this information to *p2*.

7.1.4. Extended Dataflow Diagram Visualisation

To visualise extended dataflow diagrams, ArchSec implements a visual graph editor. Since ArchSec builds on top of the Eclipse Modelling Framework (EMF), the editor is integrated into the Eclipse tool platform. Figure 7.8 depicts the editor showing the extended dataflow diagram introduced in Figure 7.3.

Figure 7.8. Visualisation of the extended dataflow diagram introduced in Figure 7.3



The right-hand side of the depiction presents the diagram in a tree-like editor, while the left-hand side shows the graphical diagram editor. The tree-like editor shows all objects from Figure 7.3, whilst the graphical editor focuses on all elements, channels, and trust areas. The latter ones are not contained in the example and therefore not visible. The editor's drawing palette adapts to the schema of the opened extended dataflow diagram. Moreover, the editor chooses icons for the depiction of elements. These icons can easily be customised to support a non-built-in schema.

The graphical editor enables the manual creation of extended dataflow diagrams and the visualisation of diagrams extracted from an application's source code. The information not visible in the graphical representation, such as data and attributes, can be edited with additional tabular editors.

7.1.5. Summary

This section explained extended dataflow diagrams introduced by and used in ArchSec. Compared to traditional dataflow diagrams, the most significant changes are a schema that is adaptable to different use cases and attributes. The attributes can assign semantics to the elements of dataflow diagrams and allow non-security experts to create extended dataflow diagrams with implicitly attributed elements by using ArchSec's predefined schema. Using an attributed extended dataflow diagram is the foundation for the later introduced automatic security flaw detection. With the graphical editor, it is also possible to visualise the diagrams and use them for a manual security assessment.

7.2. Converting System Models to Extended Dataflow Diagrams

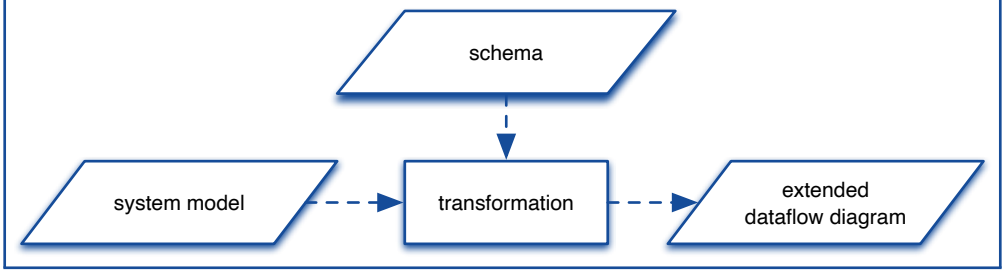
Chapter 6 introduced ArchSec's system model and explained how it extracts system models for existing applications using static analysis. The beginning of this chapter introduced extended dataflow diagrams, a more expressive form of the dataflow diagrams used in Microsoft's threat modelling. Now, this section illustrates how ArchSec bridges the gap between these two representations.

Both representations, ArchSec's intermediate representation and the extended dataflow diagram, are models in the sense of model-driven software development. Consequently, converting from a system model to an extended dataflow diagram is a model-to-model transformation. ArchSec employs the QVT language [120] for specifying the transformation between these two representations. QVT contains an imperative language part that enables mapping source model elements to target model elements using mapping functions. The complete transformation then applies to an instance of the source model and generates an instance of the target model. During the mapping process, the QVT interpreter keeps track of the mapped elements and allows one to look up target elements based on the used mapping function. With the help of this feature, it is possible to create references in the target model.

Figure 7.9 shows the input and the output of the EDFD conversion process. It uses a generated system model and an EDFD schema for generating an extended dataflow diagram. The schema is created according to the performed analyses in the extraction process.[1] The transformation itself is extensible. Listing 7.3 presents the main transformation, which maps from the `core` model

[1] ArchSec uses Eclipse's extension-point mechanism to describe the dependencies between existing static analyses, transformations and the required schema extensions.

Figure 7.9. System model to extended dataflow diagram conversion process



(cf. Section 6.2.1) to the `dfd` model. The *main* function looks for an instance of type `System` in the source model and maps it to the `DataFlowDiagram` (please compare Section 7.1.1) element from the target model. While doing so, it maps each system's `device` using the `toElement` mapping. The listing skips the remainder of the core transformation.

Listing 7.3. ArchSec's main transformation

```
import system.core;

modeltype core uses "http://informatik.uni-bremen.de/st/system/core";
modeltype dfd uses "http://informatik.uni-bremen.de/st/dfd";

transformation transform(in source : core, inout target : dfd);

main() {
  input := source.objects();
  dfd := source.rootObjects()![DataFlowDiagram];

  dfd->map toDFD(source.rootObjects()![core::System]);
}

mapping inout DataFlowDiagram::toDFD(in system : core::System) {
  elements += system.devices->map toElement();
}
```

Listing 7.4 shows the concrete `toElement`-mapping implementation for instances of the `Device` class. It converts a device to an extended dataflow diagram element of with the element type *device* and copies the *device*'s name. All `processes` stored in the `device` are mapped to elements and assigned to the generated element's list of `children`. If the `device` stores some analysis

Listing 7.4. Excerpt of ArchSec's core package's transformation

```
modeltype core uses "http://informatik.uni-bremen.de/st/system/core";
modeltype dfd uses "http://informatik.uni-bremen.de/st/dfd";

library core;

mapping core::Device::toElement() : Element {
  type := device();
  name := self.name;
  children += self.processes->map toElement();

  self.analysisInformation->map mapAnalysisInfo();
}

mapping inout core::AnalysisInformation::mapAnalysisInfo() {
  assert warning (true) with log('Unsupported AnalysisInformation of type
  ↪  ' + self.toString());
}

query core::device() : ElementType {
  return dfd.schema.getElementType("device");
}
```

information, they are mapped using the `mapAnalysisInfo` mapping. The default implementation issues a log entry stating that the concrete mapping is missing. It is necessary to overwrite the core mappings, such as the `toElement` or `mapAnalysisInformation` mapping, for any subclasses to customise the resulting extended dataflow diagram.

Finally, Listing 7.5 shows an excerpt of ArchSec's Android-specific transformation. The Android model defines an `AndroidDevice` which inherits from the core's `Device`. The transformation overwrites the `toElement`-mapping from Listing 7.4 to set an Android-specific element type. Furthermore, the mapping creates a trust area for each application that contains the components that were not exported, which other applications cannot access.

QVT allows ArchSec's system model to extended dataflow diagram transformation to be easily extensible, which was shown with a simple example from ArchSec's Android extension. By choosing transformations based on the performed static analyses and the security aspects of interest, it is possible to generate different architectural views for the threat modelling process. The resulting extended dataflow diagrams can differ in the contained information and depth of display.

Listing 7.5. Snippet of ArchSec's Android-related transformation

```
modeltype android uses
↪  "http://informatik.uni-bremen.de/st/system/framework/android";
modeltype core uses "http://informatik.uni-bremen.de/st/system/core";
modeltype dfd uses "http://informatik.uni-bremen.de/st/dfd";

library android;

mapping AndroidDevice::toElement() : Element inherits Device::toElement {
  type := device();
  diagram().trustAreas += self.processes
    ->selectByKind(android::Application)
    ->select(app | app.components
                     ->select(c | not c.exported)
                     ->notEmpty())
    ->map localTrustArea();
}

query android::device() : ElementType {
  return dfd.schema.getElementType("Device.Android");
}
```

7.3. Formal Extended Dataflow Diagram

It is possible to see the above introduced extended dataflow diagram as a graph structure. The first important thing to note is that attributes and types (c.f. Listing 7.1) are an important feature of the presented extended dataflow diagrams and therefore need to be represented in a formal definition.

Ehrig et al. define *attributed typed graphs*. Additionally, they formalise hierarchies for types [66]. Nevertheless, they do not combine these definitions. The remainder of this section gives a combined definition of both.

Definition 7 (hierarchically typed attributed graph) *A typed graph is a tuple* $G = (V, E, s : E \to V, t : E \to V)$ *with a type graph* $T_G = (T_V, T_E)$ *containing all possible types for nodes* V *and edges* E*, and a graph morphism* $type : G \to T_G$*.*

Additionally, each specific type set is equipped with its own hierarchy graph: $HG_i = (T_i, E_i), i \in \{V, E\}$*, where* $E_i \subset T_i \times T_i$ *is a partial order, making* $\mathfrak{G} = (G, T_G, type, HG_V, HG_E)$ *a hierarchically typed graph.*

Furthermore, let V_A *be a set of attribute values, as well as* E_{VA} *with* $s_{VA} : E_{VA} \to V, t_{VA} : E_{VA} \to V_A$ *and* E_{EA} *with* $s_{EA} : E_{EA} \to E, t_{EA} : E_{EA} \to$

V_A sets of edges connecting nodes V and edges E with their attributes V_A given through their respective source and target functions $s_j, t_j, j \in \{VA, EA\}$. Finally, a hierarchically typed attributed graph is defined as the tuple $\mathfrak{G} = (G, T_G, type, HG_V, HG_E, V_A, E_j, s_j, t_j), j \in \{VA, EA\}$.

The given definition already allows representing many of the features present in the ArchSec model. Elements can be formalised as nodes, and channels as edges. Furthermore, they can have types and attributes as well, and the given types can be hierarchically structured. However, to correctly formalise ArchSec's extended data flow diagram model, an inclusion of *data* and *trust areas* is required. Both can be represented as nodes with a different set of types for each. As data is ArchSec's way of modelling the flow of assets through the system, nodes that represent data need to be connected to elements and channels. To differentiate nodes representing elements, data and trust area, they receive specific sets, named V_E, V_D, V_T. Analoguesly, to differentiate edges that represent channels and those that connect elements, data, and trust areas, they are named E_C for channels, and $E_{contains}, E_{processes}, E_{includes}$. Attributes are represented as nodes as well, and to capture the value they take, a set $V_{attributes}$ contains all present attribute values, and therefore allows the specification of edges determining the value of the attribute.

Additionally, types are represented by a type graph, which the main graph is mapped to using a graph morphism, therefore specifiying a type for each element, data, trust area, channel, and attribute. These types can furthermore be hierarchically structured, which is expressed through hierarchy graphs as mentioned in the definition above.

Taking all the considerations above into account, formal extended dataflow diagrams are given by the following definition.

Definition 8 (formal extended dataflow diagram) *A formal extended dataflow diagram is a tuple $\mathfrak{FEDFD} = (V, E, \mathcal{S}, \mathcal{T}, TV, type, HG_i, V_{Attributes})$, where $i \in \{E, C, D, T, Attributes\}$ and*

$$V = V_E \cup V_D \cup V_T \cup AV_i$$
$$E = E_C \cup E_{contains} \cup E_{includes} \cup E_{processes} \cup \bigcup_{j \in V_{Attributes}} E_j.$$

E_j underlies the following condition:

$$\begin{aligned} E_j \subset & V_E \times AV_E \\ & \cup E_C \times AV_C \\ & \cup V_D \times AV_D \\ & \cup V_T \times AV_T \end{aligned}$$

Furthermore, $\mathcal{S} = \{s, s_{contains}, s_{includes}, s_{processes}\}$ denotes all source functions and $\mathcal{T} = \{t, t_{contains}, t_{includes}, t_{processes}\}$ all target functions, each of them given as follows:

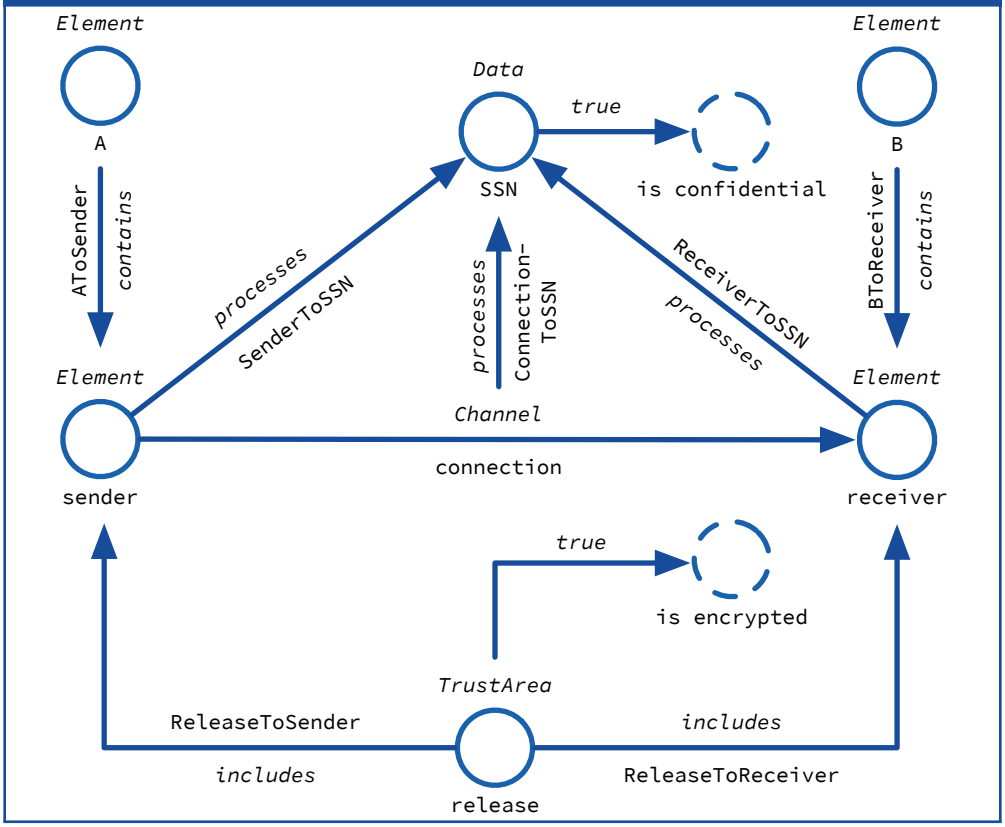
$$\begin{aligned} s &: E_C \rightarrow V_E \\ t &: E_C \rightarrow V_E \\ s_{contains} &: E_{contains} \rightarrow V_E \\ t_{contains} &: E_{contains} \rightarrow V_E \\ s_{includes} &: E_{includes} \rightarrow V_T \\ t_{includes} &: E_{includes} \rightarrow V_E \\ s_{processes} &: E_{processes} \rightarrow V_E \cup E_C \\ t_{processes} &: E_{processes} \rightarrow V_D \end{aligned}$$

$TV = (TV_E \cup TV_D \cup TV_T \cup TV_{Attributes}, TE_C)$ is a type graph with its graph morphism $type : (V_E \cup V_D \cup V_T, E_C) \rightarrow TV$, where $type(V_E) \subset TV_E$, $type(V_D) \subset TV_D$ and $type(V_T) \subset TV_T$. HG_i denote hierarchy graphs (as defined in Defintion 7) for each kind of type.

To illustrate the above shown definition, the running example is given in its formal definition. The first six elements are defined through the basic parts of the exemplary system but note that, for illustrative reasons, a trust area around sender and receiver has been introduced. Functions are given as pairs, where the first element is mapped to the second.

$$\begin{aligned} V_E &= \{A, B, sender, receiver\} \\ V_D &= \{SSN\} \\ V_T &= \{release\} \\ E_C &= \{connection\} \\ s &= \{(connection, sender)\} \\ t &= \{(connection, receiver)\} \end{aligned}$$

Figure 7.10. FEDFD - main graph (V, E, s, t, AV_i)



Next, the graph receives more edges to connect the *data* and *trust area* nodes to *elements* or *channels*:

$$
\begin{aligned}
E_{includes} &= \{ReleaseToSender, ReleaseToReceiver\} \\
E_{processes} &= \{SenderToSSN, ReceiverToSSN, ConnectionToSSN\} \\
s_{includes} &= \{(ReleaseToSender, release), (ReleaseToReceiver, release)\} \\
t_{includes} &= \{(ReleaseToSender, sender), (ReleaseToReceiver, receiver)\} \\
s_{processes} &= \{(SenderToSSN, SSN), (ReceiverToSSN, SSN), \\
&= (ConnectionToSSN, SSN)\} \\
t_{processes} &= \{(SenderToSSN, sender), (ReceiverToSSN, receiver), \\
&= (ConnectionToSSN, connection)\}
\end{aligned}
$$

Still, elements A and sender as well as B and *receiver* are unconnected, although *sender* and *receiver* belong to the elements A and B. They are

therefore equipped with *contains* edges:

$$\begin{aligned} E_{contains} &= \{ATo Sender, BTo Receiver\} \\ s_{contains} &= \{(ATo Sender, A), (BTo Receiver, B)\} \\ t_{contains} &= \{(ATo Sender, sender), (BTo Receiver, receiver)\} \end{aligned}$$

Nodes may be equipped with attributes. The attributes vary depending on *type*, in this case, the channel *connection* has the attribute *is encrypted: true*, and the data *personal information* has the attribute *is confidential: true*. Both are of type *Boolean* but this will be shown when introducing the type graph. As both attributes only have the value *true*, $V_{Attributes} = \{true\}$. Therefore, AV_i is given as below, and the required edges E_{true} as well.

$$\begin{aligned} AV_E &= \emptyset \\ AV_C &= \{is\ encrypted\} \\ AV_D &= \{is\ confidential\} \\ AV_T &= \emptyset \\ E_{true} &= \{(personal\ information, is\ confidential), (connection, is\ encrypted)\} \end{aligned}$$

To allow typing for the running example, a type graph and a graph morphism are necessary. Please note that the graph morphism is given as list of pairs, where the presence of (x, y) can be interpreted as x maps to y:

$$\begin{aligned} TV = &(\{server, process, device\} \cup \{personal\ information\} \cup \{system\} \\ &\cup \{Boolean\}, \\ &\{HTTPS = V_E \times V_E, HTTP = V_E \times V_E, \\ &\text{inter-process communication} = V_E \times V_E\}) \\ type = &\{(A, server), (B, server), (sender, process), (receiver, process), \\ &(connection, HTTPS), (release, system), (SSN, personal\ information), \\ &(is\ encrypted, Boolean), (is\ confidential, Boolean)\} \end{aligned}$$

As *type* is required to be a graph morphism, a quick check shows:

$$\begin{aligned} type(s(connection)) = type(sender) &= process \\ &= s(HTTPS) = s(type(connection)) \\ type(t(connection)) = type(receiver) &= process \\ &= t(HTTPS) = t(type(connection)) \end{aligned}$$

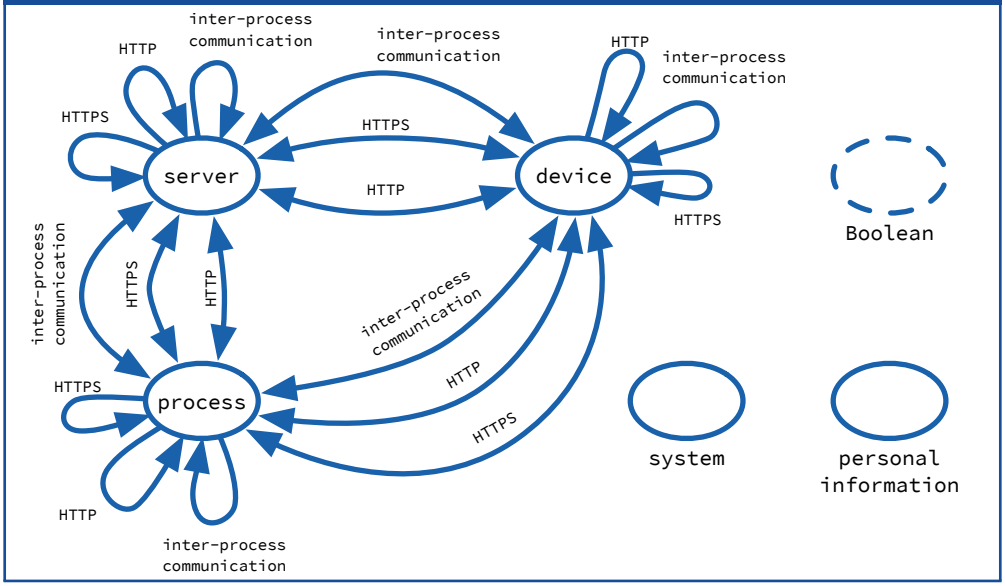
As $E_C = \{connection\}$, this proves that *type* is a graph morphism.

Hierarchical types require hierarchy graphs for every type. They are given in the standard $HG = (V, E)$ notation. Indices E, C, D, T indicate the part the hierarchy graph is built for.

$$
\begin{aligned}
HG_E &= (\{server, processes, device\}, \{(device, server\}) \\
HG_C &= (\{\text{inter-process communication}, HTTP, HTTPS\}, \\
&\qquad \{(\text{inter-process communication}, HTTP), (HTTP, HTTPS)\}) \\
HG_D &= (\{personal\ information\}, \emptyset) \\
HG_T &= (\{system\}, \emptyset) \\
HG_{Attributes} &= (\{Boolean\}, \emptyset)
\end{aligned}
$$

Figure 7.11 depicts the type graph (please note, the dashed circles represent the types for attributes) as it can be derived from this example, or the schema definition in Listing 7.1 with an additional type for trust areas. Figure 7.10 shows the main part of the $\mathfrak{FEDFD}$ as formalised in this example and given in Figure 7.3. Again, a trust area is added to show all elements of an EDFD. The hierarchy graphs are not given as figure, as they can be rather trivially derived from the formal definition given above.

Figure 7.11. FEDFD - type graph TV



Now, a $\mathfrak{FEDFD}$ is said to *describe* a program under a given schema, when the following conditions hold:

Type graph For every element, data and trust area type from the schema, there is a node in TV. Furthermore, for every channel type, there is an edge from every element type node to every element type node. For every attribute type, there is a node in TV.

Hierarchy graph Additionally, for every element, data, trust area, channel and attribute type there is a node in their respective hierarchy graph, and for every children entry in the schema, there is an edge connecting the parent with the child.

Main elements For every component, method, and parameter resulting from the component-based callgraph, there is a node in V_E, every call is represented through a channel in E_C, whose source and target are set to the caller and callee nodes. Every asset being present at an element or being transmitted through a channel, has a corresponding node in V_D and all configured trust areas are elements of V_T. This represents the main parts of the program.

Structure For every data node in V_D, there is one edge in $E_{processes}$ for each element this asset is present at and each channel that transmits this data. For every trust area node in V_T, there is one edge in $E_{includes}$ for every element that is included in this trust area. For every element that stems from another element, e.g., a method from a component, there is an edge in $E_{contains}$. This represents the internal structure of the program.

Attributes For every attribute type defined in the schema and present in the program, there is node in AV_i. For every value, an attribute takes, there is an element in $V_{attributes}$. For every channel, element, trust area, or data that is of a type which implies an attribute as defined by the schema, there is an edge in E_j, connecting the channel, element, trust area, or data with the attribute node, where $j \in V_{attributes}$ is the value the attribute takes.

The first two conditions are only dependent on the schema. The third and fourth conditions then show the structure und content of the analysed program. The fifth condition is at the intersection of the program and the schema.

7.4. Chapter Summary

This chapter introduced extended dataflow diagrams, ArchSec's means to represent architectural views containing security-related information. Extended dataflow diagrams are an extension to traditional dataflow diagrams with three main advantages. First, the schema that is part of extended dataflow

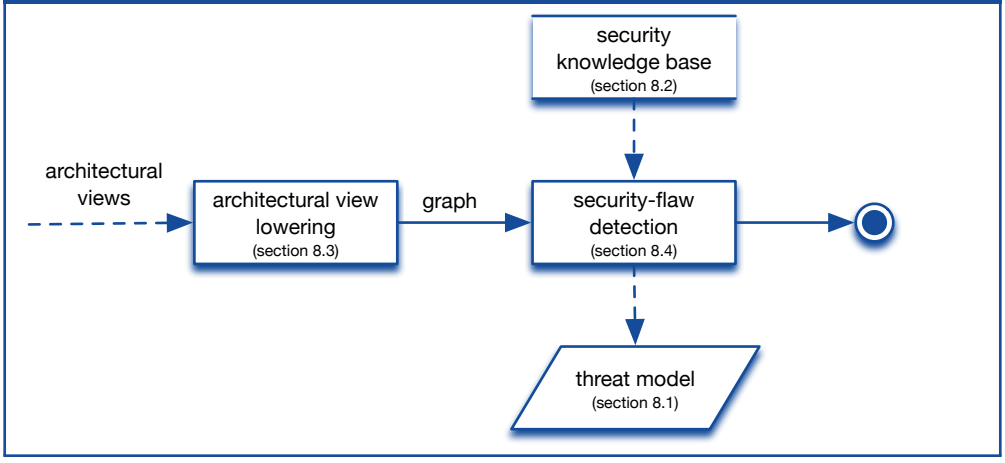
diagrams makes them very flexible and allows one to adapt them to different usage scenarios. Second, the attributes allow describing security requirements and security measures. Lastly, the implied attributes stated in the EDFD's schema give laypeople an easy step into threat modelling while reducing the security knowledge they need to have. Furthermore, this chapter described the system models' transformation into extended dataflow diagrams using a QVT-based model-to-model transformation. ArchSec can create extended dataflow diagrams with the extraction step, but identifying security flaws is still open and will be explained in the next chapter.

Chapter 8

Security Flaw Detection

So far, the Theme described the extraction of system models from a software's static artefacts and the transformation of system models to extended dataflow diagrams. To actually gain knowledge from these steps, it is necessary to derive a detection resulting in a list of security flaws. Therefore, this chapter investigates *RQ5—How can we describe architectural security flaws for dataflow diagrams* and *RQ6—How can instances of the architectural security flaw descriptions be automatically identified?* Hence, this chapter introduces ArchSec's knowledge base, which contains security flaw patterns and mitigation patterns. It describes the detection from security flaws in extracted or manually created extended dataflow diagrams. The detection process takes an architectural view and a security knowledge base as parameters. It searches for instances of the patterns stored in the knowledge base and followingly creates a threat model. Figure 8.1 gives an overview of the detection process used in ArchSec.

Figure 8.1. Outline of the security flaw detection



Section 8.1's description of the resulting threat model lays the foundation for the knowledge base that Section 8.2 introduces. Next, Section 8.3 illustrates

the lowering of extended dataflow diagrams to a simple graph structure, while Section 8.4 shows the actual detection process. Different publications have described parts of this chapter's contents or given examples for the knowledge base's content [35–39].

8.1. Threat Model

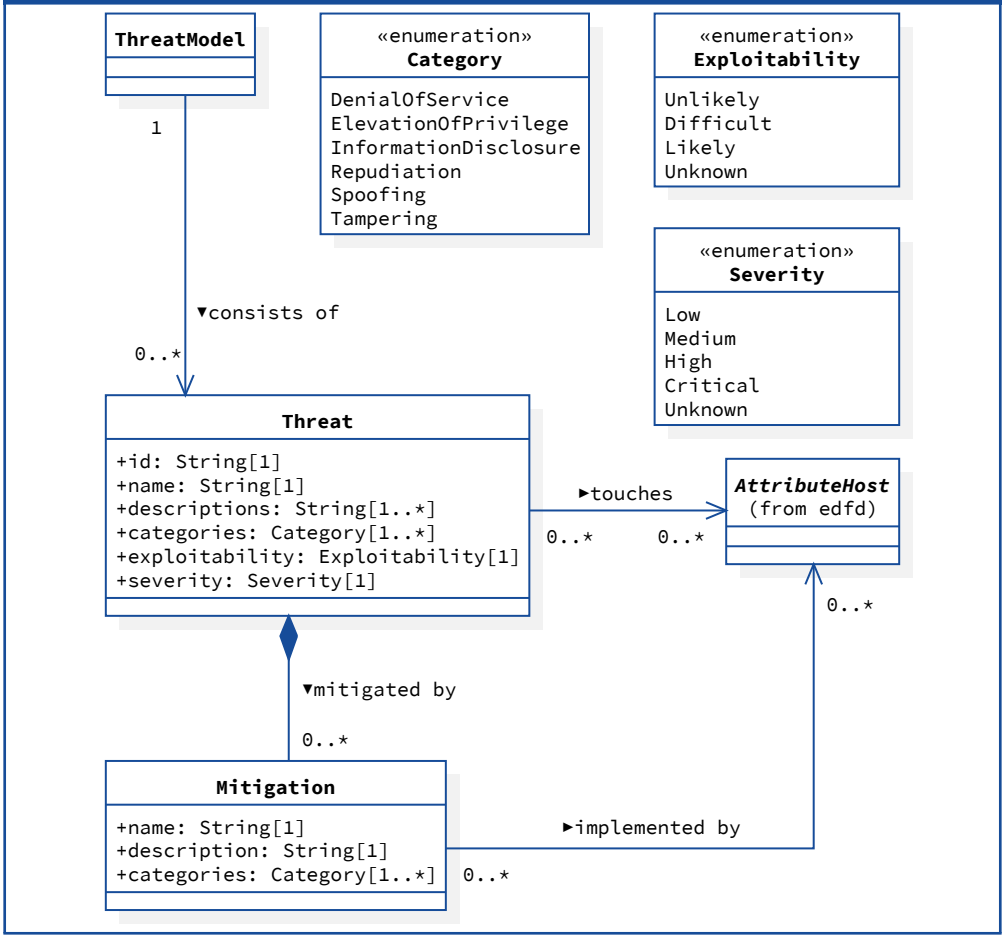
Microsoft's threat model process's resulting threat model is a list of possible threats to a system's security requirements. For each of the threat model's entries, the process collects different information. Besides describing the threat, the process tries to assess the threat's risk by categorising it. According to the threat's risk value and possibly existing mitigations, possible next steps are then planned.

Figure 8.2 shows ArchSec's version of the threat model. A `ThreatModel` consists of an arbitrary number of `Threat`s, each of them described by a unique `identifier`, a `name`, and degrees of `exploitability` and `severity`. Furthermore, a threat contains some `descriptions` and belongs to `categories`. A threat `touches` attribute hosts from the extended dataflow diagram, which can be channels, data, elements, or trust areas, describing the extended dataflow diagram elements affected by the threat. Additionally, a threat can be `mitigated by` mitigations. A mitigation has a `name`, a `description`, and several `categories`. Each mitigation is `implemented by` an attribute host, as well. The `categories` of a threat describe which protection goals the threat endangers, while the `categories` of a mitigation explain which protection goals are protected. The ordinal classification of a threat's `exploitability` and `severity` help to assess the risk of the threat.

In literature, there are different frameworks to estimate the risk of a security flaw (or a threat to security). According to ISO/IEC Guide 73 [3], risk estimation is the *process used to assign values to the probability and consequences of a risk*. The risk estimation can *consider costs, benefits, the concerns of stakeholders or other variables*. ArchSec uses a simple approach to risk estimation and categorises threats regarding exploitability and severity ordinally. Exploitability describes the likelihood of a security flaw to be utilised, while severity represents the flaw's impact.

Figure 8.3 shows an exemplary risk matrix for the categorisation of threats. Risk matrices have a long history and are, for instance, standardised in the military sector [2]. A risk matrix compares the probability of occurrence and the impact of a threat and orders the entries according to their importance. According to the idea of risk matrices, threats that are likely and have a critical

Figure 8.2. Model of the threat model

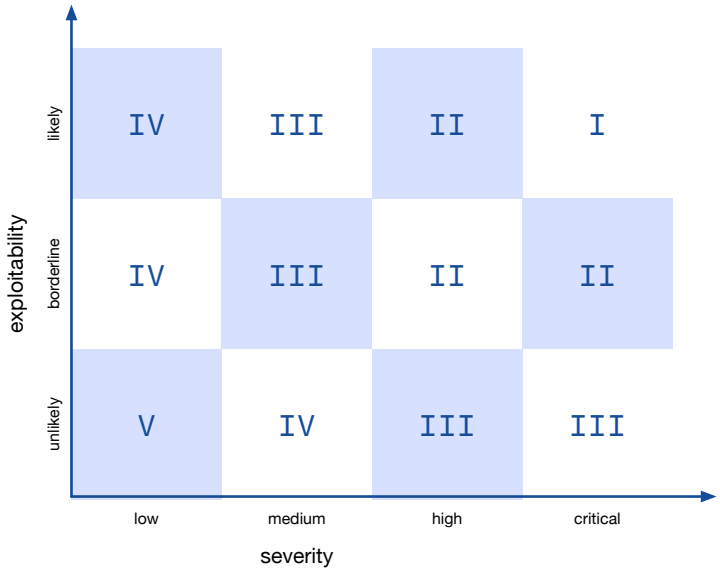


severity have to be tackled first.

8.2. Security Knowledge Base

Threat modelling uses the terms *threat* and *mitigation* to describe situations in which a system is vulnerable to an attack and ways to mitigate this attack. Appropriately, the knowledge base captures possible threat patterns and mitigation patterns. These patterns can be compared to architectural security anti-patterns, which describe circumstances that indicate a system's insecurity. Security patterns, in turn, correspond to mitigations, making a system more secure again. Since extended dataflow diagrams are graphs, it is essen-

Figure 8.3. Risk matrix for identified threats based on MIL-STD-882C [2]

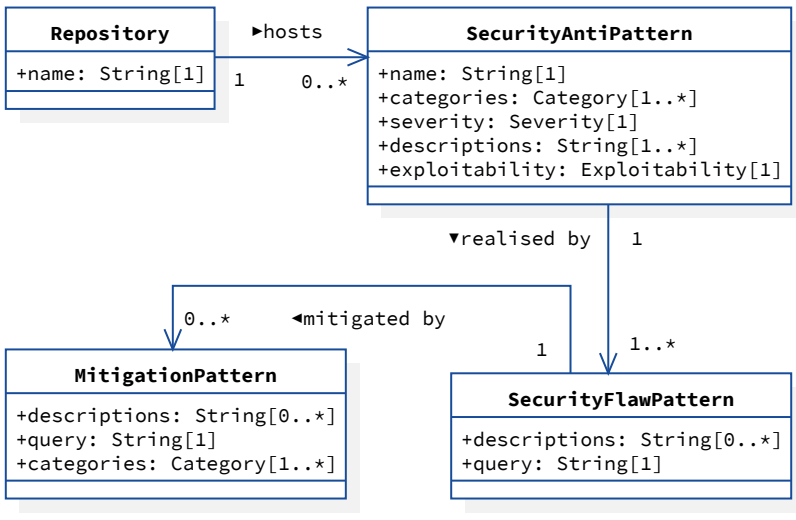


tially necessary to describe subgraphs representing an insecure system state and again—based on these subgraphs— subsequent subgraphs that describe a secure system state. The knowledge base collects these subgraph patterns for threats and mitigations alongside additional information.

Figure 8.4 shows ArchSec's knowledge base model. A `Repository` `hosts` multiple `SecurityAntiPatterns`. An anti-pattern has a `name` and several `descriptions`. Furthermore, it threatens different `categories` of protection goals and has an estimated `severity` and `exploitability`. A positive number of `SecurityFlawPatterns` realises a security anti-pattern. This modeling allows different realisations of the same security anti-pattern. A security anti-pattern has additional `descriptions`, has an associated `query`, and is mitigated by a number of `MitigationPatterns`. In turn, the mitigation patterns have `descriptions`, `categories`, and a `query`, as well. Please note that Section 8.4 will highlight the specific characteristics of the mentioned queries.

The protection goal categories a mitigation protects may only be a subset of the protection goal categories the security anti-pattern threatens since it is not intended that the employed mechanism addresses all protection goals. Therefore, a security anti-pattern is only mitigated if all threatened protection goals are covered by a mitigation. The estimated exploitability and severity are only rough estimations and are needed for a review once a concrete threat

Figure 8.4. ArchSec's knowledge base model



model has been generated.

8.3. Lowering Extended Dataflow Diagrams

The extended dataflow diagrams are lowered to more abstract graph structures on the way to security flaw detection. Concretely, ArchSec uses a directed, typed, attributed graph. Over the time of writing and with the advent of graph-based databases, this structure has become known by the name *property graph* [73]. Each node and each edge in this graph has a type and a set of attributes. Again, the transformation between the two representations is a model-to-model transformation.

Figure 8.5 shows an enhanced extended dataflow diagram and the lowered graph of the example presented in Figure 7.3. The example has been refined by two process nodes, named *sender* and *receiber*, which exchange the sensitive *SSN*. Since they are part of the same software system, they are part of the same trust area, indicated by the dashed circles.

The transformation created a node for each *element*, *data*, and *trust zone* in the EDFD. The associated `label` describes if the node resulted from an *element*, *data*, or *trust zone*. Furthermore, it maps hierarchical information, such as an *element*'s children or a *trust area*'s *elements*, to edges that are either labelled with `contains` or `includes`. An edge with the label `processes` relates processed *data* to the responsible *element*. If an extended dataflow

Figure 8.5. Exemplary lowering of an extended dataflow diagram

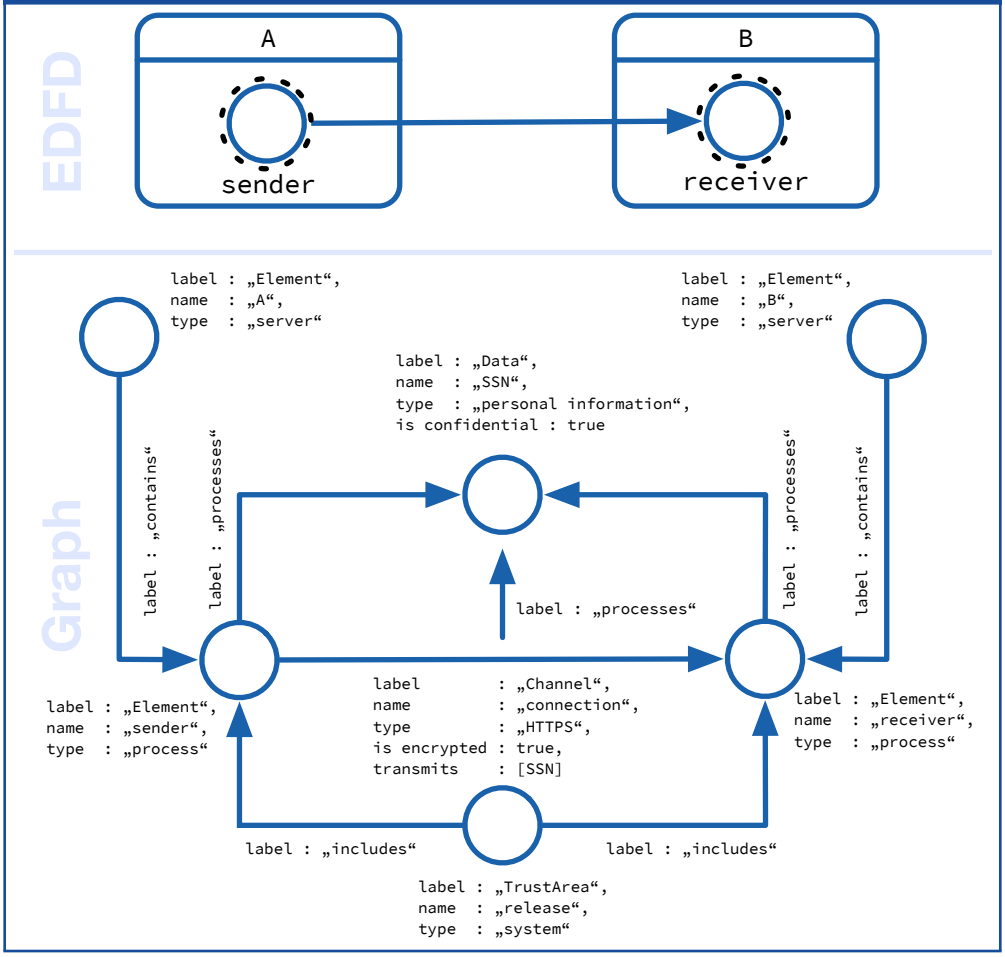


diagram member has an associated name, the name is mapped to an attribute as well. Furthermore, there are attributes for the member's type and all further attributes stored in the EDFD. A *channel* maps to a set of edges depending on the `InFlow`s and `OutFlow`s. A channel between two elements, for instance, can result in two edges. The first one represents the client's requests to the server, while the second one represents the server's responses. This makes an expression of `InFlow`s and `OutFlow`s of the extended dataflow diagram model possible.

Lowering an extended dataflow diagram can result in a very large graph. The graph is neither visualised within ArchSec nor used for purposes other than

automatic security flaw detection. Its only purpose is the detection of potential security flaws. Therefore, the size is only relevant for the detection performance.

Concluding, it is noticeable that a lowered extended dataflow diagram is close to the formalised extended dataflow diagram introduced in Section 7.3. However, the graphical visualisation is a little bit more compact, while the underlying information is the same. Some of the information, such as the type, are visualised as attributes as well as the real attributes themselves.

8.4. Automatic Security Flaw Detection

Following the lowering of extended dataflow diagrams to property graphs, the search for security flaw patterns and mitigation patterns reduces to a search for subgraphs in a graph. This problem is known as the subgraph isomorphism problem, and it is NP-complete [53].

One application of graphs and subgraph isomorphism are graph databases. Their purpose is to store graph-structured data and to query the graph efficiently. Different graph query languages have been proposed by academia and industry. An overview of graph query languages is given in Subsection 8.4.1. Afterwards, Subsection 8.4.2 illustrates queries for a security flaw pattern and for corresponding mitigations. Finally, Subsection 8.4.3 concludes this section.

8.4.1. Graph Query Languages

Graph query languages depend on the employed graph structure and usually allow one to query and manipulate the specific graph structure. Therefore, what all languages have in common is that they let the user define matching graphs. One advantage of graph query languages over traditional query languages, such as SQL, is that they allow specifying *Kleene* closures to match paths in the graph with an arbitrary length. Although this feature is part of newer versions of the SQL standard, not all SQL-based databases support this feature. Since graph databases are quite young and only became available 15 years ago, various graph languages still exist that will be briefly discussed in this subsection.

It is necessary to specify the requirements a language has to meet in order to be able to capture all necessary information of security flaw patterns and mitigation patterns to make an informed choice. A security flaw pattern is a subgraph within an extended dataflow diagram that expresses threats to the system's security. Accordingly, the first requirement for the language is the possibility to query subgraphs of a graph. The second requirement stems from the purpose of dataflow diagrams. Their task is to express the flow of information within a system. Hence, the possibility to query paths of unpredictable length within a

graph is indisputable. The last requirement results from the extended dataflows structure. An extended dataflow diagram stores the security knowledge and requirements within the assigned attributes of *channels*, *data*, *elements*, and *trust areas*. This adds the possibility to create queries respecting and returning the graph's attributes to the list.

The *SparQL* 1.1 standard combines several different standards of the W3C consortium *that provides languages and protocols to query and manipulate RDF graph content on the Web or in an RDF store* [167]. RDF, standing for resource description framework, is a means to semantically describe the information stored in web resources and their relations. According to the query language definition, *SparQL* has several limitations concerning the matching and filtering of paths in an RDF graph. In *SparQL* it is, for instance, impossible to query paths and path lengths [18]. Consequently, it is unsuitable for detecting security flaws in extended dataflow diagrams.

Gremlin is a graph traversal machine and a graph traversal language developed by the Apache Foundation [133]. *Gremlin* aims to provide a database-agnostic query language enabling developers to query different graph databases with the same language. In contrast to most query languages, *Gremlin* does not define a textual query language but a programming language dependent API for creating queries. Hence, it is rather inconvenient to use the *Gremlin* language to express and store security flaw patterns in a knowledge base. For employing Gremlin it is necessary to create a query language, such as, a domain-specific language to specify the queries which then can be translated to *Gremlin* queries, while checking a graph. Furthermore, *Gremlin* only allows querying nodes and paths but not to query values or subgraphs [18], which is necessary for the given purpose.

Cypher is a graph query language that has its origins in the *Neo4j* graph database [18]. The *Neo4j* database stores property graphs, and consequently, *Cypher* is well-suited to query such graphs and resembles SQL. A query consists of three different clauses. First, the matching clause describes a matching pattern. Second, the filter clause can reduce the number of matches based on specific properties and, finally, the return clause, is able to return nodes, paths, subgraphs, or concrete attribute values. Furthermore, *Cypher* allows the specification of paths with an arbitrary length. Therefore, *Cypher* is a good candidate for describing security flaw patterns and mitigation patterns in ArchSec's knowledge base.

In 2015, the *Cypher* language was turned into an open graph query language named *opencypher*, providing a complete grammar of *opencypher* for tool vendors to implement. Today, different open source projects use the *opencypher* language for querying graphs. With graph databases maturing over time, in

2018, the international standardisation organisation started to standardise a query language for property graphs that reuses many syntactic elements of *opencypher*. The name of this new language, specifically designed for querying and manipulating property graphs, is *GQL*, an abbreviation of *graph query language*.

This subsection introduced three different graph query languages and explained their advantages and disadvantages. Comparing these languages showed that *Cypher* is the only language that fulfils the list of requirements set up at the beginning of this subsection and can express queries for nodes, edges, paths of arbitrary length, subgraphs, and values. Therefore, it is the only language that fits the automated security flaw detection needs in ArchSec.

8.4.2. Running Example

After identifying *Cypher* as the best candidate for describing security flaw patterns and mitigation patterns, this subsection shows the definition of exemplary patterns, using *Cypher* for a better understanding. Therefore, the remainder of this subsection uses the exemplary extended dataflow diagram and its lowering shown in Figure 8.5.

Listing 8.1. Security flaw pattern description for an information disclosure flaw

```
MATCH (src : Element)
   -[host: Channel * 1 {type : subtypeof("inter-process
↪  communication")}]->
  (tgt : Element)
WHERE ANY (d IN host.data WHERE d."is confidential" = true)
RETURN src, host, tgt
```

Listing 8.1 shows the base pattern for different kinds of information disclosure flaws. The query pattern consists of two nodes, each denoted by parentheses and a connecting edge. The first node has the label `Element` and is bound to the name `src`. The second node is, as well, labelled as an `Element` and is bound to the name `tgt`, whilst the connecting edge's label is `Channel`. In this pattern, the length of the path is limited to one and the path is named host. Furthermore, the type of the edge must be *inter-process communication* or a subtype. Next, the `WHERE`-clause of the query filters possible matches and only keeps matches where the channel transfers confidential data. The query then returns the source node, the path, and the target node for each match of the

pattern.

Applied to the lowered graph shown in Figure 8.5, ArchSec's checker identifies a single match for the security flaw pattern. It maps the node `src` to the element named *server*, the node `tgt` to the element named *client*, and the path to the channel connecting the matched elements. The match survives the filtering clause because the channel transmits the confidential *SSN*.

Listing 8.2. Mitigation pattern for the security flaw pattern shown in Figure 8.1

```
RETURN ANY (a IN host.attributes WHERE a.key.subtypeof("is encrypted") AND
↪  a.value) AS mitigated
```

Listing 8.2 shows a valid mitigation pattern for matches of the previously introduced security flaw pattern. ArchSec's checker framework evaluates the query against each finding of a security flaw pattern. Therefore, it is possible to use the bound variable names from the security flaw pattern. The mitigation pattern shown in Listing 8.1 checks if the path has an assigned attribute of type *is encrypted* (or a corresponding subtype) and returns the result of this boolean expression under the name `mitigated`.

Applied to the match described above, this mitigation pattern will return `true` since the used channel type *HTTPS* implies that the channel attribute type *is encrypted* is set to `true` (please compare to the schema definition shown in Listing 7.1). Consequently, ArchSec will assume the information disclosure threat as mitigated.

Listing 8.3. Additional mitigation pattern for the security flaw pattern shown in Figure 8.1

```
MATCH (src : Element)
   <-[:contains * 0 .. ]-
  (:Element)
   <-[:includes * 1 .. ]-
  (area: "Trust Area" {type : subtype("network"), "is trustworthy" :
↪  true})
   -[:includes * 1 .. ]->
  (:Element)
   -[:contains * 0 .. ]->
  (tgt : Element)
RETURN true AS mitigated, area
```

Listing 8.3 presents another possible mitigation to the security flaw pattern based on a more complicated mitigation pattern. In essence, the pattern checks if the `source` and `target` node are in the same *trust area*, which must be of type *network*. The hierarchical nature of *elements* and *trust areas* complicates the query. Therefore, the pattern starts at the node `src` and transitively follows incoming edges labelled with `contains`. The query reaches all parent nodes of `src` using this path. Since its allowed length is zero the unnamed node can match `src` itself. Next, the query follows all incoming edges labelled `includes` for each identified element and searches for nodes labelled with `Trust Area` that have an assigned type of a subtype *network* and is marked *trustworthy*. The remainder of the rule checks if the `tgt` node is reachable from the identified trust area node.

The main idea of this mitigation is the idea that the processes exchanging confidential information run in a trustworthy network where the security analyst does not expect an attacker to control any device and therefore cannot eavesdrop on the unprotected transmitted sensitive information.

8.4.3. Summary

This section explained the equivalence of the security flaw detection problem to the subgraph isomorphism problem, which is NP-complete. Notwithstanding, there are concepts to tackle this problem resulting in graph databases. Due to their differences from traditional relational databases, they require new query languages. This section presented some of the newer graph query languages and discussed their advantages and disadvantages regarding the security flaw detection problem. Using the Theme's running example, the chosen query language *Cypher* demonstrates its applicability and capability to address the given problem.

8.5. Chapter Summary

Security flaw detection is the missing piece on the way towards a fully automated threat modelling process. A step towards an automated detection process is the enquiry of a security flaw knowledge base containing security flaw patterns and mitigation patterns, which this chapter introduced. This establishes the hypothesis for *"RQ5—How can we describe architectural security flaws for dataflow diagrams?"*. After making these patterns explicit, it is possible to search for their instances automatically. This chapter showed how ArchSec maps the pattern instance identification to the well-known subgraph isomorphism problem, offering a solution for *"RQ6—How can instances of the architectural*

security flaw descriptions be automatically identified?" The collection of all identified instances builds up an application's threat model. Following the introduced risk matrix, the threats contained in the threat model receive a ranking. The ranked threats offer security experts the necessary information to assess the overall security of a system and prioritise and plan further steps for securing, if necessary.

Part III

Variations

Chapter 9

Research Hypotheses

The Theme presented research questions that are the subject of this thesis. Moreover, the Theme already described approaches to solving these research questions and discussed these solutions in detail. These solutions divide themselves into two categories: Those where a theoretical answer is sufficient and those that need an empirical evaluation. Section 9.1 summarises the discussions from the last chapters and lists the questions that received a theoretic answer. Afterwards, Section 9.2 deals with the remaining questions and develops research hypotheses to evaluate them empirically. Finally, Section 9.3 summarises the findings of this chapter and relates them to the remainder of the variations.

9.1. Research Question Discussion

Chapter 5 identified three interesting research topics: a higher expressiveness for dataflow diagrams, the automatic extraction of architectural security views, and the automatic detection of security flaws. For these three topics, it formulated six specific research questions which this thesis aims to answer.

Considering the first topic, an unlimited number of possibilities arises to deal with *"RQ1—How can dataflow diagrams be enhanced to support a more formal expressiveness?"* Section 7.1 presented one possible approach with introducing extended dataflow diagrams to deal with this research question. Extended dataflow diagrams distinguish themselves from traditional dataflow diagrams through their formal definition and the possibility of including semantic meaning through attributes. Thus, a more formal expressiveness of extended dataflow diagram can be assumed since, up until now, traditional dataflow diagrams do not support this [63]. Still, whether this increased formal expressiveness serves its actual purpose remains to be seen: enabling an automatic extraction of architectural security flaws. Therefore, this question can only be answered entirely in correspondence with *RQ4*, *RQ5*, and *RQ6*.

Section 6.1 discussed *"RQ2—What kind of architectural information and security information are of interest during threat modelling?"* It identified relevant

architectural information and security information based on the fundamental concepts of threat modelling introduced in Section 3.3. Architectural components, entry points, and dataflows have been identified as important information that is crucial to include in dataflow diagrams to ensure successful threat modelling. Regarding security information of interest, Section 2.1 answered this research question in part, as several security topics originate in information security. The entirety of information resulting from this investigation gives the list of requirements for an automatic extraction process. Whether it is possible to extract all of this information remains a question for *RQ3* and *RQ4*.

Furthermore, Section 6.1 took a look at the research questions *"RQ3—What kind of architectural information can be automatically extracted for existing software systems?"* and *"RQ4—Which architectural security information can be extracted using static software analysis?"* The investigation identified modern application frameworks, such as JavaEE or Android, as promising candidates for extraction since these frameworks prescribe specific architecture patterns that explicitly specify possible entry points, architectural components and means for interprocess communication. Furthermore, such frameworks offer security-related application programming interfaces for authorisation, authentication, or encryption, as suggested by information security. Therefore, Section 6.3 proposed an approach to extracting such facts for existing component-based software systems based on static analysis techniques. Nevertheless, an empirical evaluation for this approach is necessary to answer *RQ3* and *RQ4* conclusively.

Turning to the automatic detection of security flaws, *"RQ5—How can we describe architectural security flaws for dataflow diagrams?"* and *"RQ6—How can instances of the architectural security flaw descriptions be automatically identified?"* work hand in hand, as the form of description influences the automatic detection and, in turn, the possible detection methods influence the chosen description. Therefore, *RQ5* received a twofold answer: On the one hand, Section 8.2 proposed a semi-formal description of architectural security flaws to be stored in a knowledge base. This knowledge base stores information on security flaw patterns, mitigation patterns, their relation to the STRIDE categories and initial risk estimation. On the other hand, every security anti-pattern and mitigation pattern obtained a formal definition in the form of a graph query, which was the result of discussing possible solutions for *RQ6*. Of course, other solutions to *RQ5* are possible, but in the light of the more formal direction of this thesis' research, a graph query language satisfied most conditions and therefore was chosen. *RQ6* found itself confronted with the semi-formal descriptions from Section 8.2 and the automatically extracted EDFDs. Section 8.3 and 8.4 developed a possible detection flow: First, the EDFD is lowered to a property graph, making it addressable through graph query

languages. Second, the semi-formal descriptions from Section 8.2 are given in queries in the chosen graph query language Cypher. Third, the lowered graph is queried for the subgraph defined in the query. An empirical evaluation is most certainly necessary to corroborate this proposition on an automatic security flaw detection flow. If it proved successful, the proposed automatic proposed detection flow would serve as an answer for *RQ6*.

Table 9.1. Discussed research questions

Name	*Question*	*Section*	*Evaluation*
RQ1	*How can dataflow diagrams be enhanced to support a more formal expressiveness?*	7.1	
RQ2	*What kind of architectural information and security information is of interest during threat modelling?*	6.1	
RQ3	*What kind of architectural information can be automatically extracted for existing software systems?*	6.1	✓
RQ4	*Which architectural security information can be automatically extracted using static software analysis?*	6.1	✓
RQ5	*How can we describe architectural security flaws for dataflow diagrams?*	8.2, 8.4	
RQ6	*How can instances of the architectural security flaw descriptions be automatically identified?*	8.3, 8.4	✓

Table 9.1 summarises the research questions, the sections that discussed possible solutions and marks those questions that require empirical evaluation to show the ArchSec approach's suitability by a checkmark.

9.2. Research Hypotheses Development

The discussion in the preceding section clearly showed that *RQ3*, *RQ4*, and *RQ6* require an empirical evaluation to support the proposed approaches for automatic view extraction and automatic security flaw detection as reasonable solutions. As the ArchSec approach divides into two main steps (extraction

and detection), it is only sensible to derive hypotheses that work alongside this deviation. Nevertheless, extraction results aiming to detect security flaws become essential in the detection phase and should therefore be evaluated fittingly. The ArchSec approach presented in the Theme claims to be able to extract the required information from *RQ2* into an extended dataflow diagram. To conclusively answer *RQ3*, the first research hypotheses can be formulated as such: *"H1—The presented static analysis techniques can correctly extract extended dataflow diagrams for component-based software systems automatically"*. Please note the restriction to component-based software systems, as these have been identified in Chapter 5 to be promising candidates for an automatic extraction process. To capture potential differences relating to the different component-based software systems, JavaEE and Android, that ArchSec is equipped to handle at the moment, a formulation of two subordinate hypotheses is necessary. The hypotheses *"H1.1—The presented static analysis techniques can correctly extract architectural views for JavaEE systems automatically"* and *"H1.2—The presented static analysis techniques can correctly extract architectural views for Android systems automatically"* serve, if evaluated positively, as good supporting pillars for an answer of *RQ3*.

The second step in the ArchSec approach concerns itself with the detection of architectural security flaws. It has two research questions connected to it: *RQ4* and *RQ6*. *RQ4* asks for the potential information that was able to be extracted in the extraction step of ArchSec, while *RQ6* asks how an automatic detection is possible. Therefore, the second research hypothesis aims at the detection step as automatic detection will necessarily show which information elements will have been extracted beforehand: *"H2.2—The proposed approach can automatically detect present authorisation-related security flaws in extended dataflow diagrams"*. The answer for *RQ2* revealed three main security information types that are of interest during threat modelling. As these different types might show different behaviour during evaluation, another partition is in order. Similarly to *H1*, *H2* splits into four subordinate hypotheses, three of them focusing on authentication-, authorisation- and cryptography-related security flaws, and the fourth serves as a mixed hypothesis for more minor security issues not related to the big three. Namely, these hypotheses are *"H2.1—The proposed approach can automatically detect present authentication-related security flaws in extended dataflow diagrams"*, *"H2.2—The proposed approach can automatically detect present authorisation-related security flaws in extended dataflow diagrams"*, *"H2.3—The proposed approach can automatically detect present cryptography-related security flaws in extended dataflow diagrams"*, and *"H2.4—The proposed approach can automatically detect present security flaws in extended dataflow diagrams that do not fit into the other categories"*. Please note

Table 9.2. Research hypotheses

Name	Hypothesis	RQ	Section
H1	*The presented static analysis techniques can correctly extract extended dataflow diagrams for component-based software systems automatically.*	*RQ3*	
H1.1	*The presented static analysis techniques can correctly extract architectural views for JavaEE systems automatically .*		10.1, 10.2, 10.3
H1.2	*The presented static analysis techniques can correctly extract architectural views for Android systems automatically.*		11.1, 11.2, 11.3
H2	*The proposed approach can automatically detect present security flaws in extended dataflow diagrams*	*RQ4, RQ6*	
H2.1	*The proposed approach can automatically detect present authentication-related security flaws in extended dataflow diagrams.*		12.1, 12.3
H2.2	*The proposed approach can automatically detect present authorisation-related security flaws in extended dataflow diagrams.*		10.1, 11.1, 11.2, 12.1, 12.2
H2.3	*The proposed approach can automatically detect present cryptography-related security flaws in extended dataflow diagrams.*		10.3, 11.3, 12.1, 12.2, 12.3
H2.4	*The proposed approach can automatically detect present security flaws in extended dataflow diagrams that do not fit into the other categories.*		10.2, 11.3, 12.1, 12.3

that the questions are formulated in a way that they are applicable for manually created extended dataflow diagrams and extracted extended dataflow diagrams. In the later case a positive example means that the proposed approach is able to detect the mentioned security flaw in the implementation of the program.

A positive evaluation of these hypotheses will give strong evidence that the proposed approach for detecting security flaws is valid. Furthermore, the differentiation of the second main hypothesis will show which architectural security information the extraction process can unearth, consequently answering *RQ4*.

9.3. Chapter Summary

This chapter summarised the current state of answers to the research questions posed in Section 5. It differentiated the research questions into those that received a qualitative answer and those that need an empirical evaluation. The qualitatively answered research questions lay the foundation for the evaluation of the remaining questions. Theme yielded the theoretical basis for practically solving the upbrought questions. Table 9.2 condenses the discussion from Section 9.2 and lists the resulting research hypotheses and the sections in Variations that will answer them.

Chapter 10

Java Enterprise

This chapter applies the ArchSec approach to JavaEE-based software systems and utilises different case studies to answer the research hypotheses introduced in the last chapter. This chapter roots in several academic publications [35, 37, 147] that described the extraction process, knowledge base rules, and the resulting extended dataflow diagrams used for JavaEE-based systems. Consequently, this chapter reproduces these publications and describes some aspects in greater detail. The following sections reference these publications.

First, Section 10.1 presents an approach to checking access control policies for JavaEE-based systems. The method shown in the paper assumes the usage of JavaEE's built-in RBAC-based access control concept. Published in 2010 [147], it is the first work to lay out the initial ideas of ArchSec. Therefore, the original article used a predecessor of extended dataflow diagrams, namely resource flow graphs [128], and the publication's contents have been replicated for this thesis using extended dataflow diagrams.

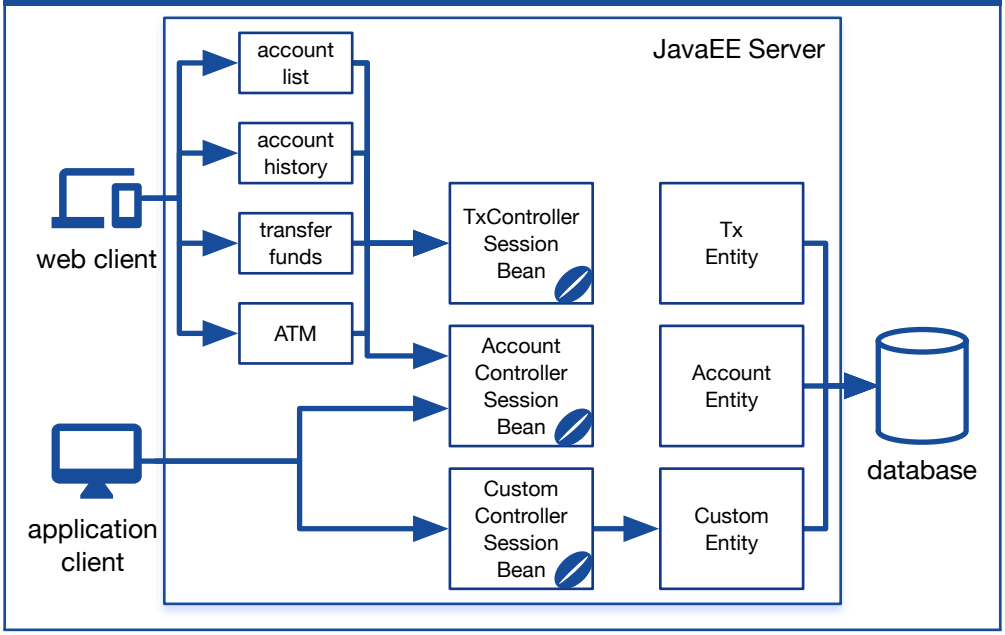
10.1. Checking Access Control for JavaEE

The RBAC model implemented in JavaEE allows assigning users to different roles. Based on these roles, developers can restrict access to specific program functions or web pages. Therefore, the implemented model resembles an access control matrix and not an RBAC model (please c.f. Section 2.2). The following case study focuses on the research hypothesis *"H2.2—The proposed approach can automatically detect present authorisation-related security flaws in extended dataflow diagrams"*. It searches for differences between the planned access control policy and the implemented access control policy to contribute to the validation of hypothesis *H2.2*. If there is a difference between these two, it can be assumed that a user of the system can access information he is not meant to access. Consequently, the system under investigation contains a security flaw.

In JavaEE, there are different possibilities to check whether a subject is assigned to a specific role. As already mentioned, it is possible to restrict access

to program functions and web pages. Access to web pages can be restricted using XML-based configuration files. However, this was not the focus of this case study. Instead, it took a look at the restrictions to program functions. Therefore, there are two possibilities: First, there are API methods, such as `HttpServletRequest.isUserInRole(String)`, that check whether the currently interacting user is in the specified role. Second, beginning with JavaEE 5, it is possible to use annotations, such as `RolesAllowed`, to declare the roles required to call the annotated methods. Identification and extraction of policy enforcement points are necessary to compare the allowed control flow to search for non-allowed accesses in a given software system. Therefore, the planned access control policy is required for this comparison, which is part of the software's architecture.

Figure 10.1. Intended architecture of *Duke's Bank* according to Oracle [87]



Duke's Bank gives an illustration of JavaEE-based acccess control, as it is a tutorial banking application provided by Oracle allowing clerks to administer customer accounts and customers to access their account histories and perform transactions [87]. The implementation provides a web-based client for customers and a rich client user interface for the clerks. In total, it consists of three different JavaEE projects, with a total of 5300 lines of application code. This does not

respect the size of additional framework- and container-specific libraries used by *Duke's Bank* that were part of the analysis. Figure 10.1 shows the architecture sketch given in the tutorial's documentation. The main idea is that a customer can access his account via the web interface, whereas clerks can only use the rich client interface. *Enterprise Java Beans* provide the functionality, also known as business logic, of the application. Namely, these *Enterprise Java Beans* are called `AccountControllerSessionBean`, `TxControllerSessionBean`, and `CustomerControllerSessionBean`. These beans encapsulate access to the database containing the account data using the *Java Persistence API*, a programming interface for declaratively mapping Java objects to SQL-based databases.

Based on the architecture sketch shown in Figure 10.1, it is possible to derive the access control policy given in Table 10.1. According to this policy, *customers* are allowed to call methods in the `TxControllerSessionBean`, *clerks* may call the methods of the `CustomControllerSessionBean`, and both roles are permitted to call methods in the `AccountControllerSessionBean`.

Table 10.1. Access control matrix for *Duke's Bank* derived from Figure 10.1

Object	**Subject**	
	clerk	**customer**
`TxControllerSessionBean`		*call*
`CustomControllerSessionBean`	*call*	
`AccountControllerSessionBean`	*call*	*call*

Duke's Bank uses the `RolesAllowed` annotation to specify the roles required for accessing the Enterprise Java Beans. Therefore, an analysis extension identifies such annotated classes and methods and enhances the Enterprise Java Beans' methods with additional information on role requirements. The resulting extended dataflow diagram elements representing these methods have additional attributes listing the required roles. Based on the policy defined in Table 10.2, queries can be written that identify possible violations. Table 10.2 shows the general rule description stored in ArchSec's knowledge base.

Listing 10.1 shows the Cypher query for identifying all Enterprise Java Bean methods within a JavaEE system that belong to the rule given in Table 10.1. It searches for elements of type `Method` contained in a class annotated by either `Stateful` or `Stateless` since this is the correct way of declaring an Enterprise Java Bean. Next, Listing 10.2 and Listing 10.3 show mitigation rules that check two rows from the access control matrix. The third mitigation rule is the same

Table 10.2. ArchSec rule for detecting improperly protected Enterprise Java Bean methods

Rule	Improper protected business method		
Severity	`High`	**Exploitability**	`Likely`

Categories	`Elevation of privilege`
	`Information disclosure`
	`Spoofing`
	`Tampering`		

Description	Enterprise Java Beans usually host a JavaEE's business logic that requires proper protection. If the implementation fails to successfully protect the business logic properly, a system's user may access information or functionality he is not allowed to access. Consequently, he might be able to elevate his privileges, to access sensitive information, hide his identity, or tamper with information stored in the system.		

Listing 10.1. Cypher query for detecting Enterprise Java Bean methods

```
MATCH
  (method : Element {type : "Method"})
    <-[:contains *1]-
  (class  : Element {type : subtypeof("Type")})
    -[*1]->
  (annotation : Element {type : "Annotation"})
WHERE annotation.attributes.linkname IN
  ["javax.ejb.Stateless", "javax.ejb.Stateful"]
RETURN method AS host
```

as the one shown in Listing 10.2 but with a different link name and role name. A link name in this case, is a unique identifier for certains programming language entities, such as, classes, attributes, or methods, and their format is specified by Java's Virtual Machine Specification [111]. The mitigation checks whether the matched class is the `TxControllerSessionBean`. Please note that the link names have been shortened for the thesis and readability reasons. The security flaw is mitigated if the attribute that holds the required roles only contains the role name `bankCustomer`. Thus, the flaw is not mitigated if there are more, fewer or other roles required. The second mitigation checks if both roles are

allowed for `AccountControllerSessionBean` methods by checking the link name of the class and the required roles.

Listing 10.2. Cypher query for detecting correctly annotated methods in `TxControllerSessionBean`

```
WHERE class.attributes.linkname = "TxControllerSessionBean"
RETURN method.attributes.requiredroles = ["bankCustomer"] AS mitigated
```

Listing 10.3. Cypher query for detecting correctly annotated methods in `AccountControllerSessionBean`

```
WHERE class.attributes.linkname = "AccountControllerSessionBean"
RETURN method.attributes.requiredroles = ["bankAdmin", "bankCustomer"] AS
↪  mitigated
```

Applying a knowledge base containing this single application-specific rule with three mitigations rules to the extended dataflow diagram yields two concrete methods which are not protected according to the intended authorisation policy. First, the method `TxControllerSessionBean.getDetails(...)` is accessible for customers and clerks instead of only being accessible for customers. Second, `CustomControllerSessionbean.getDetails(...)` is accessible for both roles as well instead of being accessible for clerks only. Whether these findings need to be addressed cannot be quickly concluded from the software's source code. However, there are arguments that these findings would be problematic and why they are not acceptable. Thus, this judgement would be the result of further risk estimation

A manual assessment of the results showed the correctness of the findings. Thus, this analysis identified differences between the planned access control policy and the implemented one, even in this simple tutorial application. These results show the necessity of such checks for automatically verifying the security architecture conformance. While the rules are hand-written and the mitigation patterns are application-specific, it is possible to automate this task. It is possible to automatically generate a knowledge base storing a JavaEE's authorisation policy.

With the presented results, this case study supports research hypotheses *H1.1* and *H2.2*. Using ArchSec, it was possible to extract an extended dataflow diagram for a JavaEE system and identify authorisation flaws. While

the application code is relatively small in size compared to real-world systems, it has the typical structure of Java enterprise applications. Since the approach was only tested with a single software system, this demonstration points towards the validity of the hypotheses, but further case studies are necessary to strengthen this reasoning. Nevertheless, it can be assumed that the approach is transferable to other applications since *Duke's Bank* is an industrial reference implementation created by Oracle—the company that, for a long time, steered the development of the Java platform, Enterprise Edition—and it showcases the possibilities of the JavaEE framework. Therefore, it is likely that other industrial applications are structured similarly.

10.2. Inter-Session Data Flow

This section deals with the possible security implications stemming from the use of the object pool pattern. Kircher and Jain introduced the object-pool pattern in 2002 [100]. The idea is to reuse resources, such as network connections or instances that are, in some way, expensive to obtain. For example, in the case of a network connection, the relevant cost factor is the time necessary to establish a new connection to another machine. While establishing a new connection only takes a split second, this duration is a very long time concerning modern computers' computing speed. Accordingly, reusing already established connections can improve the processing time significantly.

Different JavaEE frameworks and even JavaEE itself specify components that might be subject to object pooling. The already introduced stateless Enterprise Java Bean, for instance, is reused by the application container for serving different requests. Their specification states: *"Since stateless session bean instances are typically pooled, the time of the client's invocation of the create method need not have any direct relationship to the container's invocation of the* `PostConstruct/ejbCreate` *method on the stateless session bean instance."* [59, 60, 137]. This formulation does not prescribe the behaviour of an application container but gives a hint that application container implementations may use the object-pooling mechanism.

A *Java Servlet* is a managed entity that reacts to requests and produces some response returned to the request's origin. The *Servlet API* and the lifecycle that a *Servlet container* has to provide are defined in the *Java Servlet* specification. An example of a *Servlet* is the `HttpServlet` interface that responds to HTTP queries, and an implementing class can generate any valid HTTP response. Additionally, the specification mentions that the servlet container may choose to pool such objects [50].

JavaServer Pages (JSP) are means to easily generate dynamic web pages, which are a little bit outdated nowadays since newer alternatives are available. Nevertheless, they are still in use. A programmer can write a textual file which he enhances with some quoted Java code to insert dynamic data at runtime. A typical field of application for JSPs is the generation of HTML and XML documents, which are handed to a web browser. JSP files are compiled into Java bytecode and then executed. According to the *JavaServer Pages* specification [134], the base interface for all JSPs is `javax.servlet.jsp.HttpJspPage` which is a *Java Servlet*. Therefore, the container can pool compiled JSPs files, too.

Another major framework using the object pooling pattern is *Apache Struts*. This framework provides the model-view-controller pattern [129] for *JavaServer Pages*. The first version of this framework added *actions* to take the role of controllers. An action is a Java class that inherits from `org.apache.struts.action.Action`. The online documentation of *Struts 1* explicitly states that one should not use instance variables within actions due to possible multi-threading issues that originate from the fact that actions are reused to serve multiple requests.

From a performance point of view, the reuse of objects by implementing the object pooling pattern is a good idea [148], however, making an application vulnerable to security flaws. Developers using the frameworks mentioned above do not explicitly implement the pattern, but the runtime environment of their software does. None of the specifications mentioned above provide some method to clear an object's state when the framework returns it to the pool. This deficit results in problems when pooled instances store a state that is not cleared. Enterprise applications allow different users to interact simultaneously. Unique identifiers assigned to each user allow the container to distinguish different users and provide an independent session for each of them. Ideally, a separate process or memory region is assigned to each session to avoid private data leaking between sessions. Pittyfully, the object pool pattern connects sessions with each other by using the same object in different sessions. If this fact is not taken into account, it allows information to flow between different sessions.

The concept of object pooling contradicts a strict separation of sessions since it explicitly shares object instances containing sensitive information. The sharing of objects may lead to the information flow mentioned above between different sessions and, therefore, between different users. The introduced frameworks do not clear sensitive data when instances are returned to the pool.

Listing 10.4 shows a shortened exemplary inter-session data flow problem taken from an industrial application. The listing shows a *Struts action* for sending an e-mail. The action `ExampleAction` has a private attribute `recipient` of

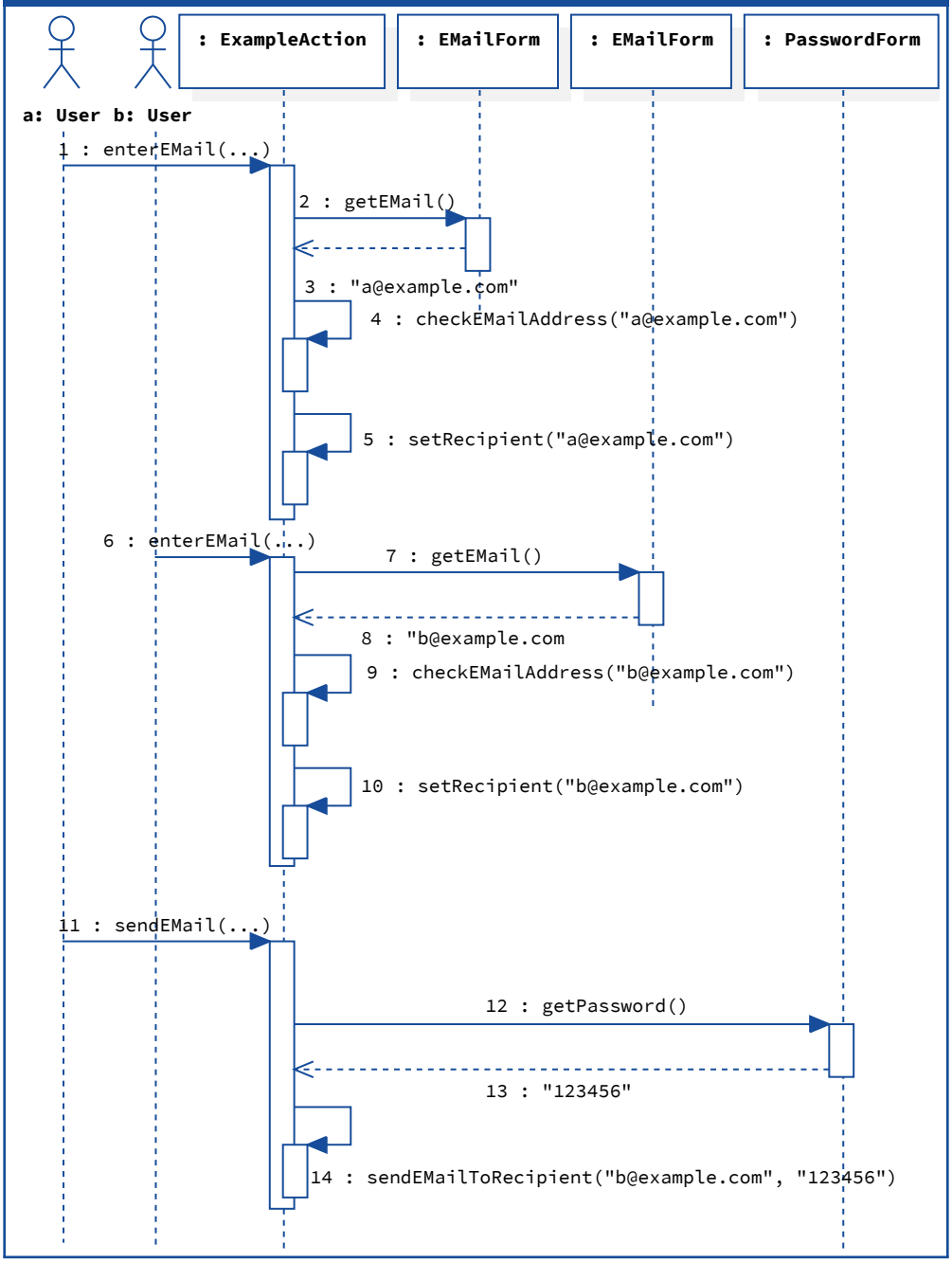
Listing 10.4. Struts action with an inter-session dataflow problem

```
// imports are ommitted

public class ExampleAction extends Action {
  private String recipient;

  private void setRecipient(String recipient) {
    this.recipient = recipient;
  }

  public ActionForward enterEMail(ActionMapping mapping,
                                  ActionForm form,
                                  HttpServletRequest request,
                                  HttpServletResponse response)
                           throws IOException, ServletException {
    EMailForm emailForm = (EMailForm)form;

    String address = emailForm.getEMail();

    if(checkEMailAddress(address)) {
        setRecipient(address);
    }
  }

  public ActionForward sendEMail(ActionMapping mapping,
                                 ActionForm form,
                                 HttpServletRequest request,
                                 HttpServletResponse response)
                           throws IOException, ServletException {

    PasswordForm passwordForm = (PasswordForm) form;

    sendEMailToRecipient(this.recipient, passwordForm.getPassword());
  }
}
```

type String, which the methods `enterEMail` and `sendEMail` access. Since these methods are *Struts actions*, they are invoked by HTTP requests, depending on the HTTP parameters. The former method stores an e-mail address provided by the query into the action's attribute and does some sanity operations. The latter method uses the stored attribute's value to send some private information to it.

While this code does not look suspicious in the first place, the class' attribute

Figure 10.2. Sequence diagram for the code example shown in Listing 10.4



will result in a security problem at runtime if two users, e.g. *Alice* and *Bob*, interact with the action simultaneously. Figure 10.2 shows a UML sequence diagram of the same piece of code. *Alice* calls `enterEMail`, and the method stores the e-mail address she entered (`a@example.com`) in the action's attribute. Next, *Bob* calls `enterEMail`, and consequently the method stores `b@example.com` in the class' attribute. Finally, *Alice* calls `sendEMail`, and the method reads the password stored in the action's parameter and sends it to the e-mail address stored in the action's attribute. Therefore, the system sends the password entered by *Alice* to the mail address entered by *Bob*.

Table 10.3. ArchSec rule for possibly pooled classes.

Rule	Misuse of pooling pattern		
Severity	`Medium`	**Exploitability**	`Likely`

Categories	`Information disclosure`
	`Spoofing`
	`Tampering`		

Description	Frameworks, such as JavaEE or Struts, can reuse pooled classes to reduce an application's memory and execution footprint. Pooled classes can pose a risk to the application's security if pooled classes use attributes to store user-specific information.		

To counteract this security-relevant problem, ArchSec has a rule for improperly used pooled instances. This purely structural pattern is not based on dataflows. The reason for this is that the dataflows as such would not be prohibited as long as they are within a session. The problem only arises from the improper use of the framework used. Table 10.3 gives the general rule description. Unfortunately, there is a different detection pattern for each presented framework. Listing 10.5 exemplarily shows the Cypher query for detecting *Struts actions* that contain attributes that are accessed. It is sufficient to have one single access in the source code to be matched since the same method can be called successively by different users. Therefore, the match of the second access is optional. Unfortunately, there are no mitigation patterns for this kind of problem. Nevertheless, the findings can be further subdivided according to the types of accesses found:

read-only: A pooled instance's attribute may be harmless if it is only read. If the attribute's type is an immutable type, such as the base types `int`, `float`, `boolean`, or a `String`, the attribute cannot lead to an inter-session dataflow.

In this case, the program's state is not manipulated and therefore, no user-specific sensitive data can flow between sessions. Otherwise, a method of the instance may be called, which changes the instance's state, possibly leading to an inter-session data flow.

write-only: It is very likely that a write-only attribute poses no threat to the protection of sensitive information. Such information may be stored in such an attribute, as there is no way it leaves the user's session and is copied into another user's session.

read and write: This category leads to inter-session dataflows. The example shown in Listing 10.4 and Figure 10.2 belongs to this class.

no access: If there is no access to a pooled class' attribute, the query shown in Listing 10.5 will not match, and, consequently, it is not marked as vulnerable.

Listing 10.5. Cypher query for detecting Struts actions with attributes

```
MATCH
  (action : Element {type : "Class", linkname :
↪  "org.apache.struts.action.Action"})
    <-[:* 1.. {type : "extends"}]-
  (class  : Element {type : "Class"})
    -[:contains *1]->
  (method1 : Element {type : "Method"})
    -[calls1 : * 0.. {type : subtypeof("Call")}]->
  (method2 : Element {type : "Method"})
    -[access1 : * 1 {type : subtypeof("Access")}]->
  (field : Element {type : "Field"}),
  (class) -[:contains *1]-> (field)
OPTIONAL MATCH
  (class)
    -[:contains *1]->
  (method3 : Element {type : "Method"})
    -[calls2 : * 0.. {type : subtypeof("Call")}]->
  (method4 : Element {type : "Method"})
    -[access2 : * 1 {type : subtypeof("Access")}]->
  (field : Element {type : "Field"})
WHERE
  access1 <> access2
RETURN method AS host
```

As already mentioned, there are no mitigations to this threat. Nevertheless, there are mitigation patterns that add additional information to the categorisa-

tion according to the categories introduced above, yielding detailed information on the finding.

The evaluation tries to find evidence if inter-session dataflows can be identified using the ArchSec approach and whether inter-session dataflows is a problem in real-world software systems. Therefore, it analyses three case-study applications, two being open source and the other being proprietary software. The criterion for selecting these applications was whether they used one of the introduced frameworks. The first analysed open-source application is the Java version of *enNode2* [6]. A running instance of *enNode2* is part of the Exchange Network, which aims to exchange environmental information between different States, Territories, Tribes, and the U.S. Environmental Protection Agency. The second application, called *GSS* [7], is a file storage service used for the Greek research and academic community.

The closed-source application has been provided by a research project's partners. It is a thriving commercial business application that helps companies to declare goods electronically for import and export. The software is offered to customers on a software-as-a-service basis, making the confidentiality of the user data an essential requirement that is challenging to ensure due to the size of the existing source code (over 600k LoC and more than 1000 JSP files).

Table 10.4 lists the different frameworks of interest used by the analysed programs. Within the table, *EJBs* stands for stateless Enterprise Java Beans, not for all types of existing Enterprise Java Beans. An interesting observation is that *enNode2* is a web application that does not use Enterprise Java Beans for its backend, whereas *GSS* does not employ the Struts framework.

Table 10.4. General information on the analysed systems

project	*EJBs*	*JSPs*	*Servlets*	*Struts 1*	*LoC [k]*	*extraction [min]*	*analysis [min]*
enNode2		✓	✓	✓	85	2:27	0:43
GSS	✓	✓	✓		36	0:24	0:07
Commercial	✓	✓	✓	✓	614	4:42	1:51

Furthermore, Table 10.4 shows the size of the analysed systems, measured with

sloccount[1], the time necessary for the extraction, and the time the analysis took. The proprietary software system is by far the largest case study application. Table 10.5 sheds light on the question whether the developers of the considered applications created pooled classes with attributes. According to the results, 24% of all implemented classes in *enNode2* are pooled. 68% of these have at least one attribute. In *GSS*, just 8% of all classes are pooled, but each of these classes contain attributes. Finally, *Commercial*'s percentage rate of pooled classes lies at 30%, and a high rate of 84% of the pooled classes contain attributes.

Table 10.5. Frequency of pooled classes

project	# classes	# pooled classes	∴ with attributes	μ(attributes)
enNode2	438	107 (24 %)	68 %	6.48
GSS	164	14 (8 %)	100 %	6.29
Commercial	4901	1485 (30 %)	84 %	8.72

enNode2 has 473 attributes in 107 pooled classes, *GSS* has 88 attributes in 14 pooled classes, and finally, *Commercial* has 10,894 attributes in 1,485 pooled classes. Therefore, inter-session dataflow seems to be a problem in real-world applications since 68% to 100% of all pooled classes have attributes. To determine whether the results may lead to inter-session dataflows, the categorisation of the attributes is relevant, which Table 10.6 shows. Most of the attributes are either only read or not accessed at all. More than 80% of the attributes found in each application belong to these two groups. An interesting fact is that the distribution differs for all three systems.

Noticeable is the high rate of unaccessed attributes within the pooled classes. There are three possible explanations for an attribute not being referenced. First, it actually might be unused. Second, it is accessed with the help of reflection, which is unlikely from a programmatic point of view. The JSP compiler transforms JSP files into Java code. During this transformation, the JSP compiler copies the Java code that is part of the JSP file directly to the resulting Java file. In this situation, there is no need of using reflection for

[1] https://dwheeler.com/sloccount/

Table 10.6. Results grouped by access type

project	read-only	write-only	read & write	no access
enNode2	54.12 %	0.42 %	4.23 %	41.23 %
GSS	30.68 %	0.00 %	0.00 %	69.32 %
Commercial	57.35 %	4.56 %	11.92 %	26.16 %

accessing the fields. Lastly, the attributes are static and final, which is an artefact of the analysis of Java bytecode instead of Java source code. Final static attributes of basic type are inlined in Java's compilation process, and therefore, the attribute seems unreferenced.

A manual inspection of 10% of each of the findings and categorisation showed no incorrectly reported or categorised results. This inspection showed that the JSP-compiler generated many of the attributes that belong to the read-only category. These attributes do not transfer data between sessions. To check wether the identified attributes of the category *read & write* can be used by an attacker to transfer data between sessions of different users, we took a closer look at the findings of the commercial software system. A maintainer of the software vendor inspected the findings and was able to identify several findings that have caused trouble in the past. In particular, customers complained that their data were messed up with data from other customers sporadically. The programmers could not identify the cause of the problem because they could not reproduce the symptoms, which the users reported. Therefore, for some of the findings, a demonstrator has been implemented for then different attributes. The demonstrator repeatedly called the specific actions until the results indicated that an *"successful"* inter-session dataflow occurred. In the end, removing the identified attributes fixed the erroneous behaviour.

Regarding the thesis' research hypothesis, this evaluation supports *H1.1* and *H2.4*. The extraction algorithm can extract extended dataflow diagrams, which are successfully employed in detecting security flaws that threaten the protection goals defined by information security.

10.3. Extracting EDFDs for JavaEE systems

A case study with two commercial JavaEE applications was conducted to additionally investigate research hypotheses *"H1.1—The presented static analysis techniques can correctly extract architectural views for JavaEE systems automatically"*. Furthermore, the case study applies a new rule related to research hypothesis *"H2.3—The proposed approach can automatically detect present*

cryptography-related security flaws in extended dataflow diagrams“.[2]

The main focus of this evaluation is a better understanding of how system experts judge the automatically extracted extended dataflow diagrams. For this purpose, an extended dataflow diagram has been extracted for each of the commercial applications. Additionally, the automatic security-flaw detection was applied to the extracted dataflow diagrams using the new ArchSec rule. In the following interview, the developers and a quality assurance representative responsible for the software under investigation first created a dataflow diagram manually. Then the extracted diagrams and the findings of the detection process were discussed with them. During the interview, the experts were asked whether the extracted diagrams correctly represent their software systems and if the found security flaws can be found in the implementation as well.

The security flaw rule introduced for this evaluation is based on the *Common Weakness Enumeration* (CWE) entry. The MITRE Corporation manages the Common Weakness Enumeration and incorporates weaknesses identified by its community [5]. The particular entry this evaluation focuses on is *CWE-5: J2EE Misconfiguration: Data Transmission Without Encryption*, which seemed suitable since both analysed systems are Java enterprise applications. Table 10.7 shows ArchSec's rule description for *CWE-5*.

Table 10.7. ArchSec rule describing *CWE-5*-related flaws

Rule	*CWE-5*: J2EE Misconfiguration: Data Transmission Without Encryption		
Severity	`High`	**Exploitability**	`Likely`

Categories	`Elevation of privilege`
	`Information disclosure`
	`Spoofing`
	`Tampering`		

Description	Information sent over a network can be compromised while in transit. An attacker may be able to read or modify the contents if the data are sent in plaintext or are weakly encrypted.		

Listing 10.6 shows the concrete cypher query for ArchSec's rule for *CWE-5*. It looks for two elements that are connected by an *inter-process communication* channel that transports *confidential* information. The corresponding mitigation pattern that is not shown here—since Listing 8.2 already showed

[2]The contents of this sections was published at CSMR'13 [37].

it—checks whether the transmitted data is encrypted. Possible used encryptions strategies—which are subtypes of the attribute type encryption—are transport or message encryption. In JavaEE, the usage of channel encryption for client communication can be requested using the application's XML-based deployment descriptor.

Listing 10.6. Cypher query for detecting *CWE-5*-related flaws

```
MATCH (src : Element)
   -[host: Channel * 1 {type : subtypeof("inter-process
↪  communication")}]->
  (tgt : Element)
WHERE ANY (d IN host.data WHERE d."is confidential" = true)
RETURN src, host, tgt
```

Based on the shown rule, the evaluation focused on the following evaluation questions:

EQ1: Does application experts agree with the automatically extracted extended dataflow diagrams from ArchSec?

EQ2: Which parts of the architecture can ArchSec extract automatically?

EQ3: Can ArchSec identify instances of the CWE-5 based security flaw?

As already mentioned, this evaluation uses two commercial case study applications. The first application is from the e-government domain and is a JavaEE-based Web Service. It is implemented using the SOA principle and consists of web services, traditional JavaEE applications, and regular Java applications. This evaluation used a subsystem responsible for creating qualified digital signatures[3] for arbitrary documents. In particular, this application makes a batch-signature mechanism available for signing many similar documents with the same key. This software contains different JavaEE component types and uses different kinds of inter-process communication means. Due to the application's task, security is a crucial aspect. The second case study is a commercial business application from the logistics domain that helps companies declare goods electronically for import and export and is used by hundreds of customers every day. The application follows the JavaEE specification in version 1.4.

[3] Qualified digital signatures are legally binding in Germany and are hence counterpart of hand-written signatures. Consequently, special measures must be put in place to secure the process of signature generation.

ArchSec extracted an extended dataflow diagram for each case study object. Then a knowledge base containing the rule introduced in Table 10.7 was applied. The generated extended dataflow diagram and the resulting threat were then inspected and discussed with application experts.

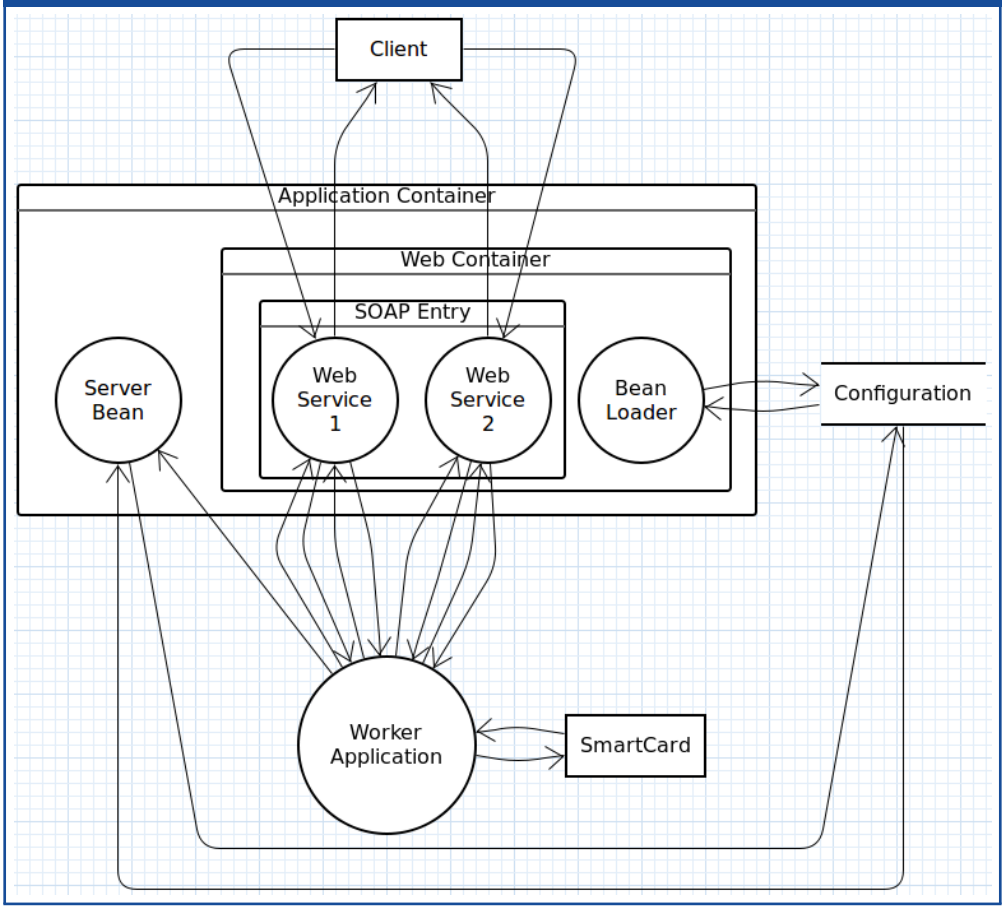
The extended dataflow diagram extracted for the first case study needed only a few refinements to satisfy the needs of the security expert. Figure 10.3 depicts it. Please note that the names of the elements are changed due to confidentiality reasons. ArchSec detected nine top-level components as well as several subcomponents that implement the provided functionality. The shown diagram does not show the subcomponents of the *Worker Application* since there are more than 20 of them. Furthermore, the subcomponents are also the cause for the multiple edges between the *Web Service* components and the worker applications. The edges target subcomponents of the Worker Application, and the shown edges are representatives of these edges. However, the inspection process removed four top-level components from the extracted EDFD, which are not part of the depiction. According to the software experts, these components are test programs that are related to third-party libraries and are thus not relevant to the expert.

The extended dataflow diagram extracted for the first case study needed only a few refinements to satisfy the needs of the security expert. Figure 10.3 depicts it. Please note that the names of the elements are changed for the sake of confidentiality. ArchSec detected nine top-level components as well as a set of subcomponents that implement the provided functionality. The inspection process removed four of the top-level detected components from the extracted EDFD because they are test programs related to third-party libraries and are thus not relevant for the expert.

Besides the structure, ArchSec also identified security measures, such as authentication based on client certificates and the use of WS-Security [117] to enable message-level security and time stamping. ArchSec could not identify vulnerabilities in the extracted extended dataflow diagram because the developers implemented mitigations for all identified threats. Thus, all identified security flaws were protected by mitigation patterns. This result is not surprising as the application has been evaluated several times according to the Common Criteria (CC), a standard for the security evaluation of IT products. Therefore, the software has been subject to several security reviews.

There were several differences between the manually created dataflow diagram and the automatically extracted one. According to the security expert, ArchSec found several processes that do not belong to the system (the test programs mentioned above related to third-party libraries). Nevertheless, these programs could be started by employees and therefore interact with the system.

Figure 10.3. Extracted data-flow diagram of the e-government application.



Furthermore, ArchSec could not detect the trust boundaries, which were then added manually to the diagram. Nevertheless, the extracted extended dataflow diagram helped to understand and manually assess the system's security aspects, leading to security-relevant components. For example, based on the extracted diagram, it was possible to manually identify an authorisation vulnerability allowing every authenticated client to use all smart cards to sign any document. The extended dataflow diagram inspection led to a *Worker Application* component responsible for enforcing authorisation decisions. The workshop's participants conducted a manual code review of this component with this knowledge at hand, which led to the security flaw.

For the second case study object, ArchSec extracted an extended data-

flow diagram with three top-level elements. First, an external entity— a web browser—communicates with the JavaEE application container using HTTP. The second entity is the JavaEE application container, which exchanges data with the third element, which is an SQL database. The application-container process subdivides into a web frontend and a business logic part according to the deployment information. These parts are further subdivided according to the framework's means they use. ArchSec detected security measures used within the software, such as authentication, and added this information to the resulting extended dataflow diagram.

The software experts' dataflow diagram was quite different to the one ArchSec extracted. Nevertheless, they agreed with the extracted one. While the software experts structured their dataflow diagram according to licensable blocks, the extraction could not automatically identify them since the code structure did not reflect this separation. Reviewing the extracted EDFD, the software experts did not identify any mistakes.

The security flaw detection identified several security flaws. ArchSec revealed that the communication between the web browser and the web frontend component transports credentials unprotected. This finding corresponds to the introduced rule for *CWE-5*. A subsequent manual inspection of the finding revealed that the developers failed to configure the framework correctly to ensure the usage of TLS over HTTP.

The conclusions of the conducted case studies are:

EQ1: The extracted extended dataflows were not identical to the ones the security experts created. Nevertheless, the security experts agreed with the diagrams after some discussion and refinement.

EQ2: ArchSec was able to identify different processes, data stores and communication channels between these objects. Unfortunately, it could not extract the trust boundaries, and the external entities needed refinement. This result is not surprising because external entities, on the one hand, are not within the analysis scope. On the other hand, Hernan et al. stated that trust boundaries are very subjective concepts [81].

EQ3: In the second case study, we successfully identified security vulnerabilities, including *CWE-5*, automatically employing a knowledge base. The software experts confirmed all identified the correctness of all identified mitigated and non-mitigated instances of this rule.

This case study, investigated research hypothesis *H1.1* and successfully identified a security flaw related to *H2.3*. This case study supports research hypotheses

H1.1. The fact that the application experts agreed with the extracted extended dataflow diagrams is an indicator that the automatic extraction was successful.

10.4. Chapter Summary

The presented JavaEE-related evaluations support different research hypotheses. All three case studies focus on identifying security flaws in extracted extended dataflow diagrams. Therefore, all presented case studies support research hypothesis *"H1.1—The presented static analysis techniques can correctly extract architectural views for JavaEE systems automatically"*. The detected security flaws belong to different categories of security flaws of research hypotheses *"H2—The proposed approach can automatically detect present security flaws in extended dataflow diagrams"* and therefore support research hypotheses *"H2.2—The proposed approach can automatically detect present authorisation-related security flaws in extended dataflow diagrams"*, *"H2.3—The proposed approach can automatically detect present cryptography-related security flaws in extended dataflow diagrams"*, and *"H2.4—The proposed approach can automatically detect present security flaws in extended dataflow diagrams that do not fit into the other categories"*.

Chapter 11

Android

This chapter utilises the ArchSec approach to analyse the Android platform and Android apps. Support for the Android platform was added to show that the proposed method is applicable for component-based systems other than JavaEE. Therefore, this chapter gathers evidence to support research hypothesis *H1*. Section 6.2.2 introduced details of the Android platform and Android's component model. This chapter reports different academic publications, which will be referenced in the following sections.

First, Section 11.1 describes a semi-manual inspection of Android's Bluetooth service, Android's Bluetooth API provider. Afterwards, Section 11.2 describes the transitivity-of-trust problem in Android application interaction. Finally, Section 11.3 shows that the ArchSec approach can analyse hybrid Android apps consisting of Java-based and HTML5-based components.

11.1. Checking Android Access Policy

The Android platform offers a permission-based access control system that implements an access control matrix. The speciality in the Android ecosystem is that the device owner assigns the permissions to the different apps. An app can request specific permissions in its deployment descriptor, such as access to the internet or the device's location. The user can decide to grant or deny these permissions. Nota bene, in earlier versions of Android, the user could only choose to accept the permissions or cancel an app's installation process.

This case study focuses on semi-automatically validating the access enforcement within the Android Bluetooth API.[1] Therefore, a security expert (not the author of this thesis) used an extracted view of the Android Bluetooth API and investigated three questions he came up with after reading the Bluetooth API's documentation. The documentation states that an application needs the `BLUETOOTH` permission to use the Bluetooth device. If an app wants to administer the Bluetooth device, it requires the `BLUETOOTH_ADMIN` permission in

[1] The contents of this paper has been published at WCRE '11 [31].

addition. The documentation explicitly notes that an app needs the former permission to use the latter. The Android documentation furthermore lists several methods, such as `enforceCallingOrSelfPermission`, as a means to enforce permissions. The first method parameter states the permission to be checked. If the app holds the permission, the method normally returns the control flow, but it throws a `SecurityException` if it does not hold the permission resulting in an interruption of the normal control flow.

For the evaluation, a detection mechanism for Android enforcement points was added to ArchSec's extraction process. This information is added to the resulting system model and then transferred to an extended dataflow diagram. The security expert was then asked to investigate the questions he had from reading the documentation. After the investigation, a retrospective think-aloud protocol [104] was created where the security expert explained his steps. The following paragraphs summarise his procedure. The expert focused on the following evaluation questions:

EQ1: Where are permissions enforced within the Bluetooth API (enforcement points)?

EQ2: Which permissions does the Bluetooth API enforce (access control policy)?

EQ3: Does the Bluetooth API enforce that a user needs the `BLUETOOTH` permission to use the `BLUETOOTH_ADMIN` permission?

Android's Bluetooth service provides the implementation of the Bluetooth API and is, consequently, the focus of this evaluation. In the beginning, the security expert wanted to know where permissions are enforced during service usage. Therefore, the security expert focused on the methods belonging to the Bluetooth service. For him, it soon became apparent from the extended dataflow diagram that all public methods call Android's enforcement methods to ensure that the calling process has the required permissions. The security expert concluded the information that all public methods are protected against unauthorised usage. This knowledge led to the next question, namely, which permissions do the enforcement points enforce?

The security expert was also interested in a view that shows the specific permissions enforced, which would visualise the service's access control policy. While he could answer the first question with the extended dataflow diagram existing so far, it was necessary, as already mentioned, to extend the extraction mechanism. This extension gathers the enforced permissions from the source code to create such a view. The relevant information for the desired view,

the permission, which will be enforced, is the first actual parameter passed to the method `enforceCallingOrSelfPermission`. In general, it is not always possible to statically determine the actual value of a method's parameter. Nevertheless, it was possible in the case of the Android platform because the compiler's constant propagation propagated the used permission literals to the method calls. The adapted extraction process adds new nodes for the checked permissions and new edges between methods and the required permissions. This adjusted view allowed the security expert to see the implemented access control policy of the Bluetooth service for the first time. Based on the generated view, he was able to identify those methods of the service belonging to the administrative part of the API. Unfortunately, Android's platform documentation did not state this information explicitly when the evaluation was conducted.

Having extracted the implemented access control policy of the Bluetooth service, the next question the security expert was interested in was whether the condition entioned in the security documentation was up-to-date, i.e., the security expert wanted to know whether all methods enforcing the `BLUETOOTH_ADMIN` permission require the `BLUETOOTH` permission, too. While it is possible to check this manually, an Android-specific rule has been created checking this requirement. Table 11.1 shows the corresponding rule description for this security requirement.

Table 11.1. ArchSec rule for improperly protected methods

Rule	Insufficient protected Bluetooth API method.		
Severity	`Medium`	**Exploitability**	`Likely`
Categories	`Spoofing`		
Description	According to the Android API every method that enforces the `BLUETOOTH_ADMIN` permission has to enforce the `BLUETOOTH` permission as well. This requirement is not enforced by the Android framework itself. Therefore, a programmer may forget to enforce the `BLUETOOTH` permission when the `BLUETOOTH_ADMIN` permission is enforced.		

While this kind of review is manually possible, an automated check is desirable. Listing 11.1 shows the Cypher rule that detects methods that enforce the `BLUETOOTH_ADMIN` permission. All these methods are prone to the problem that the programmer forgets to enforce the `BLUETOOTH` permission. This threat is mitigated if the mitigation pattern in Listing 11.2 matches.

Applying the rule to the extracted extended dataflow diagram of the Android

Listing 11.1. Cypher query for detecting methods requiring the BLUETOOTH_ADMIN permission

```
MATCH
  (perm : Element {type : "Permission", name :
↪  "android.permission.BLUETOOTH_ADMIN"})
    <-[:* 1.. {type : "requires"}]-
  (method : Element {type : "Method"})
RETURN method AS host
```

platform, ArchSec identified 14 methods that enforce the BLUETOOTH_ADMIN permission. Still, only four methods directly or indirectly check the BLUETOOTH permission as well. The remaining ten methods do not check for both permissions. Hence, the check identified differences between the documentation and the actual implementation.

Listing 11.2. Mitigation pattern checking if a method enforced the BLUETOOTH permission

```
MATCH
  (method)
    -[calls : * 0.. {type : subtypeof("Call")}]->
  (: Element {type : "Method"})
    -[:* 1.. {type : "requires"}]->
  (: Element {type : "Permission", name : "android.permission.BLUETOOTH"})
RETURN true AS mitigated
```

These findings were quite surprising for the security expert since the documentation stated that a calling process must have both permissions. Indeed, the fact that not both permissions are checked does not necessarily lead to a vulnerability. Still, a developer may develop her code with wrong assumptions in her mind after reading the documentation. Furthermore, it is necessary to review those cases where the BLUETOOTH permission is only checked indirectly if this can lead to an inconsistent state of the Bluetooth service if a calling process only has the BLUETOOTH_ADMIN permission. In summary, the results of the semi-automatic review process are the following:

EQ1: All enforcement points within the Bluetooth service were identified correctly, and it was possible to verify that all public methods were protected adequately.

EQ2: It was necessary to adapt ArchSec's extraction mechanism to extract additional information to identify the enforced permissions. Afterwards, ArchSec was able to visualise the access control policy.

EQ3: The statement that one needs the `BLUETOOTH` permission to use the `BLUETOOTH_ADMIN` could not be supported by the automatic inspection using a custom knowledge base entry since ArchSec found counterexamples.

This early case study shows that the extracted views are suitable for detecting application-specific security vulnerabilities and help security experts assess a software's security architecture manually. Therefore, it supports research hypotheses *H1.2* and *H2.2*.

11.2. Transitivity-of-trust in Android App Interaction

The second Android related evaluation focuses on the interaction of multiple apps on an Android device and the confused deputy problem introduced in Chapter 2.1 on information security. The Android platform offers a wide range of sensitive information, such as location information, pictures, and many others. Another essential concept in the Android platform is inter-app communication. Inter-app communication allows apps to outsource some functions to be solved by others. If an app, for instance, wants to edit an image, it creates an intent to edit the image and sends the intent together with the permission to access the image to an app that is capable of editing images. A malicious app can undermine this concept to access sensitive information without requesting permission at installation time. In the given example, an evil image editor app gets access to edited images without requesting the permission to access them in the first place. This way, potential dataflows are not obvious for users. If the malicious app has access to the worldwide web, it can leak the images without the user noticing.

The publication [24] presented in 2013 focused on a suitable presentation of the extracted information for Android users who are less security conscious. The aspect that is related to the thesis contents is the extraction of flow information. The identification process consists of three steps. First, an architectural view for each Android app is extracted. The view contains intra-process dataflows and information on inter-process communication. Second, the extracted views are then combined, and the targets of inter-process communications are resolved. The last step identifies information flows of sensitive information to the worldwide web. Table 11.2 shows the knowledge base rule for transitivity-of-trust problems in Android apps.

Table 11.2. ArchSec rule for detecting transitivity-of-trust problems in Android apps

Rule	Hidden information flow in Android app interaction		
Severity	`Medium`	**Exploitability**	`Medium`
Categories	`Information disclosure`		
Description	Android apps can interact with each other without requesting permissions. Some applications pass sensitive information, which are protected by permissions, to other apps using inter-process communcation means. This allows apps to silently send sensitive information to the worldwide web without the user noticing.		

Within an extended dataflow diagram, a transitivity-of-trust problem is a path of sensitive information starting in one app, crossing app boundaries to another app, and being sent to a remote service. Obviously, the path can cross more than one app boundary if the data flows through several apps. Listing 11.3 shows the corresponding Cypher query.

The query searches for an element contained in the trust area *Android platform* that processes sensitive information. Next, it looks for a communication path with an arbitrary start and length, followed by an *inter-process communication* of length one. Afterwards, there can be an arbitrary number of *inter-process communication* channels, followed by a final *inter-process communication*. All nodes within the path, except for the `tgt` node, must be contained in the same `device`. Furthermore, the source and target of the first *inter-process communication* have to be in different trust areas of type *Android app*. All channels within the matched path have to transport the sensitive information matched in the beginning.

The mentioned publication [24] focused on a feasibility study and used two apps from Google's Play store with installations on up to 50,000 devices. The first application is called *EfaFahrplan*, an interface to a public transport routing Web application, primarily improving the input form to take advantage of the context (current location and time and previous searches). *EfaFahrplan* takes parameters such as origin, destination and desired arrival time and sends a query to the routing Web application. Thus, the *EfaFahrplan* application requires unrestricted Web access privileges. When entering the public transport routing parameters, the user may choose to take the current location as the origin. Since using detailed location data is not strictly necessary for the application's primary goal, location queries have been factored into a separate

Listing 11.3. Threat pattern for the transitivity-of-trust problem

```
MATCH
  (ta1 : TrustArea {name : "Android platform"})
    -[: contains * 1..]->
  (src: Element),
    -[: processes * 1]->
  (d : Data {"is confidential" : true}),
  (src)
   -[: Channel : * 0.. {data : contains(d)}]->
  (n1: Element)
   -[: Channel {type : subtypeof("inter-process communication"), data :
↪  contains(d)} * 1]->
  (n2:Element)
   -[: Channel {type : subtypeof("intra-process communication"), data :
↪  contains(d)} * 0..]->
  (n3:Element)
   -[: Channel {type : subtypeof("inter-process communication"), data :
↪  contains(d)} * 1]->
  (tgt : Element),
  (n1) <-[: contains * 1..]- (ta2 : TrustArea {type : "Android App"}),
  (n2) <-[: contains * 1..]- (ta3 : TrustArea {type : "Android App"}),
  (device : Element {type : subtypeof("Android Device")} -[:contains *
↪  1..]-> src,
  (device) -[: contains * 1..]-> (n1),
  (device) -[: contains * 1..]-> (n2),
  (device) -[: contains * 1..]-> (n3)
WHERE
  ta2 <> ta3
  AND NOT EXISTS (tgt) <-[: contains * 1..]- (device)
RETURN d AS information, src AS source, tgt AS target, device AS host
```

Android component installed as a separate application. The second application, named *EfaQueryLocation*, requires the location permissions. With two separate applications, a user may choose whether she would like to grant location access. *EfaQueryLocation* sends the location information to *EfaFahrplan*, which sends it to the world wide web.

The evaluation focused on whether it is possible to detect such indirect information flow using static analyses and dealt with the question of how such information flows can be presented to the user. A first analysis prototype was able to extract the information for the mentioned apps and create a combined diagram. Based on this diagram, it was possible to identify the presented information flow.

The presented evaluation supports research hypothesis *H1.2* since it deals with the extraction of flow information across multiple Android apps. Additionally, it deals with the confused deputy problem, which is related to research hypothesis *H2.2*.

11.3. Hybrid Android Apps

A special kind of Android UI components is the Android web view. A web view is a web browser running in the application's context, allowing it to render HTML content that even can be loaded from a server. There is a strict security rule in traditional browser applications: A website cannot access the system's resources, such as files. This rule does not hold for web views, since depending on the configuration and the app's permissions, a web view has access to resources and even may call program and platform code. Therefore, these web views are an attractive attack vector since an attacker might inject his JavaScript code, making it easy to change an app's behaviour.[2]

Nevertheless, hybrid apps are an attractive category of apps for large software vendors. There are specialised JavaScript libraries, such as Apache Cordova, focusing on providing a common abstraction layer for different mobile phone operating systems. Such an API allows software vendors to create a single HTML5-based app and deploy it on Android, iOS, or Windows Phone, saving them time and effort to create separate apps for each of these platforms.

Listing 11.4. Example of an Android activity displaying a web view

```
import android.app.Activity;

public class MainActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        final WebView myWebView = new WebView(this);
        setContentView(myWebView);
        myWebView.loadUrl("https://www.example.com");
    }
}
```

Listing 11.4 shows a small example of an Android activity showing a web

[2]The contents of this section has already been published at D • A • CH Security 2014 [36]

view, which loads its content from the world wide web. Depending on Android's version, it is possible to configure different aspects of the web view. For instance, it is possible to use custom TLS certificates, customise the certificate checks, or enable a JavaScript-to-Java bridge. These configuration options offer great flexibility but pose the risk of security flaws.

It is necessary to examine a hybrid's Android app's Java code, HTML5 code and JavaScript code to identify the use of insecure connections. The Java code of the app configures the web view and loads an initial HTML document for rendering. Consequently, ArchSec has to parse HTML5 and JavaScript to identify resources loaded by the web view. While parsing and understanding pure HTML5 pages is not complicated, statically analysing JavaScript is challenging because of its dynamic nature. Therefore, the static analysis support is not as precise as for the Java code. Nevertheless, the analysis is able to identify a significant number of loaded URL's using an intra-procedural analysis, but it is necessary to state that the results of the implemented analysis is not complete. The loaded files can either be stored locally within the app's resources or remotely on a server. If the analysis identifies remotely loaded files, it loads these dynamically, parses them, and analyses their content.

Additionally, an analysis was implemented to classify the certificate handling of web views. There are two ways to change the behaviour of the certificate handling. First, a web view uses a class implementing `TrustManager`. This class checks whether a certificate is trustworthy. If it is not, its implementation has to throw a `SecurityException`. The analysis checks if the implementation may throw the named exception with an inter-procedural analysis. Second, a web view uses a class implementing `HostNameVerifier` to check if the name of the domain the web view communicating with matches the domain name given in the certificate. The implementation checks if the implementation only returns constant `boolean` values.

Table 11.3 shows the ArchSec rule description used for identifying vulnerable web views.

Listings 11.5 shows the Cypher query for the rule given in Table 11.3. It searches for Android activities marked as `WebView`, which have an incoming inter-process communication that starts at a resource. All findings matching this description are marked as vulnerable. The mitigation pattern, which is omitted here, checks if the detected communication channel uses proper encryption. Further extensions for detecting TLS insecurities and JavaScript bridge usage, as well as corresponding knowledge base rules, have been added as well.

The evaluation dealt with the following questions:

EQ1: Do hybrid apps load the content shown remotely via HTTP instead

Table 11.3. ArchSec rule for detecting code injection attacks for Android web views

Rule	Remote code execution		
Severity	`High`	**Exploitability**	`Likely`

Categories	`Information disclosure`
	`Spoofing`
	`Tampering`		

Description	Android web views render web pages within an Android app. This poses the risk an attack might inject JavaScript-based program code if the app loads code via an improperly secured connection.		

Listing 11.5. Cypher query for detecting web views that are vulnerable to code injection attacks

```
MATCH
  (view : Element {type : "Activity", "Android.IsWebView" : true})
    <-[channel: Channel *1 {type : subtypeof("Interprocess")}]-
  (doc : Element {type : subtypeof("Resource")})
RETURN channel AS host, view AS view, doc AS doc
```

of HTTPS?

EQ2: Do hybrid apps break TLS security by using custom certificate checks?

EQ3: Do hybrid apps enable the JavaScript bridge and allow the remotely loaded code to interact with the operating system?

In total, 938 Android apps loaded from Google's Play Store have been analysed to answer the evaluation questions. The app vendors are smaller companies and many large companies, such as Intel, SAP, Siemens, and Telekom. According to their description, these were hybrid apps. Of these 938 apps, 87 did not utilise Android's web view and, therefore, were excluded from the analysis. From the remaining 851 apps, ArchSec identified 161 apps that load content using HTTP. 77 apps implemented their certificate checking in a way that makes the HTTPS connection vulnerable to a man-in-the-middle attack where an attacker can use a self-signed certificate to hijack the communication channel. 453 of the analysed apps enabled the JavaScript bridge and allowed the JavaScript code full access to the Android platform.

Figure 11.1. Extracted EDFD for the Wikipedia Android app

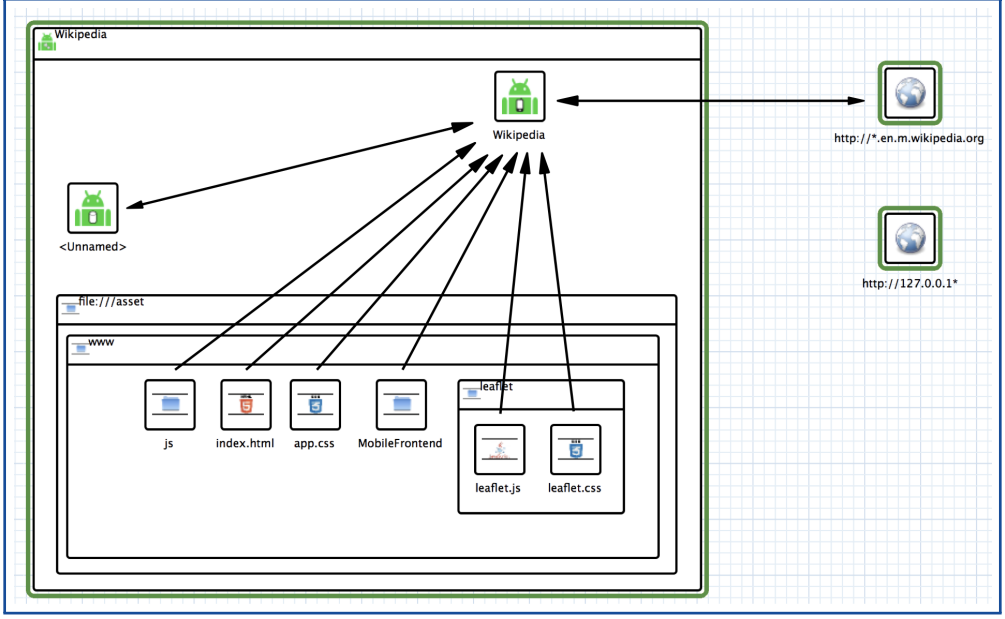


Figure 11.1 shows an extended dataflow diagram for an older version of the Wikipedia app. The app consists of two components: an unnamed content provider and an activity, a web view. The web view loads different files from the app's deployment bundle and some files from the English Wikipedia server. The trust area, depicted by a green border, stems from the application's web view configuration, limiting access to the shown domains. For loading Wikipedia articles, the app uses HTTP, allowing attackers to easily inject custom JavaScript code if they want to. The app activated the JavaScript bridge allowing them to have access to the device's geographical location. An attacker can use this feature to call other framework features, such as the microphone and camera. For this app, a feasibility setup showed that a successful attack is possible, and the geolocation is accessible and can be sent to the Wikipedia server. Since the connection is not encrypted, the attacker can extract the device's current location.

To conclude the evaluation questions, it is possible to state:

EQ1: 161 out of 851 analysed hybrid Android apps use insecure communication to load their content, making them vulnerable to code injection attacks.

EQ2: 77 out of 851 analysed hybrid Android apps break TLS security by implementing or using a broken certificate checking.

EQ3: 453 out of 851 analysed hybrid Android apps employ the JavaScript bridge and allow the JavaScript code to use framework means.

Regarding the research hypotheses, this evaluation supports research hypothesis *H1.2* by successfully extracting extended dataflow diagrams for Android-based apps. Furthermore, it even analysed JavaScript code and HTML5 code to identify possible dataflows. Moreover, this evaluation supports research hypotheses *H2.3* and *H2.4* by detecting problems in TLS configuration and code injection attacks.

11.4. Chapter Summary

This chapter focused on presenting different evaluations with Android apps. These were either complete Java-based ones or hybrid Android apps. Therefore, this chapter focused on research hypothesis *H1.2*, which all three evaluations support. In all three evaluations, architectural views were automatically extracted and used for automatic security flaw detection.

Regarding the second primary research hypothesis, this chapter gave evidence for all security-related hypotheses but *H2.1*. Section 11.1 and Section 11.2 dealt with authentication-related security flaws, therefore supporting research hypothesis *H2.2*. Section 11.3 dealt with security topics concerned with research hypotheses *H2.3* and *H2.4*. The fact that no case study for authentication-related security flaws is presented is not surprising. Android is mainly focused on single-user usage, and therefore Android deals with authentication at the platform level. Thus, no evaluation for research hypothesis *H2.1* is presented.

Chapter 12

Detection

This chapter focuses on detecting security flaws in manually created extended dataflow diagrams. The diagrams are either created manually or extracted from other models, which in turn, were created manually. Furthermore, this chapter compares ArchSec's detection with an alternative approach to automatic security flaw detection.

12.1. Extracting Threats from Extended Data Flow Diagrams

This evaluation focuses on automatic security flaw detection in manually created extended dataflow diagrams and is, therefore, mainly related to research hypothesis *H2*. For this purpose, the thesis' author created a new set of ArchSec rules that might be related to JavaEE systems. The newly created rules are based on the *Common Weakness Enumeration* and the *Common Attack Pattern Enumeration and Classification* (CAPEC). MITRE Corporation manages this list as well [4]. Table 12.1 shows the introduced rules. Besides the rule's source and the number of each rule, the table states the ranking number in the *CWE Top 25* list. The table also states whether the ArchSec rules consist only of queries for the threat (T) or of a query for the threat and corresponding mitigation rules (M). Since some of the rules are very general, it might be that the ArchSec rule cannot cover all aspects of the security flaw. If the patterns do not match all possible threat or mitigation aspects, the T or M is given in braces.[1]

For this purpose, the thesis' author and application experts manually created extended dataflow diagrams for two industrial case studies. The experts created the diagrams without the application's source code. Thus, they modelled their mental model of the applications. After applying ArchSec, the identified threats were discussed, and the application experts explained whether there are

[1]The contents of this section was published at ESSoS'16 [38].

Table 12.1. Newly introduced rules

source	number	CWE Top 25	patterns
CWE	79	1	T
CWE	89	2	T
CWE	78	3	T
CWE	306	5	M
CWE	311	8	M
CWE	352	12	T
CWE	22	13	T
CWE	327	19	M
CWE	759	25	T
CWE	288		M
CWE	319		M
CWE	602		M
CAPEC	108		T
CAPEC	16		T
CAPEC	22		T/(M)
CAPEC	66		T
CAPEC	94		(T)/(M)

existing mitigations. In some cases, they consulted the existing application's source code to present the mitigations. Each of the diagrams was created in two meetings, and the effort to construct these diagrams summed up to half a day of work. The assessed applications are both from the logistics domain having a similar purpose. In particular, the following evaluation questions were investigated:

EQ1: Is it possible to automatically identify threats and security flaws in manually created extended dataflow diagrams since they might differ from automatically extracted ones?

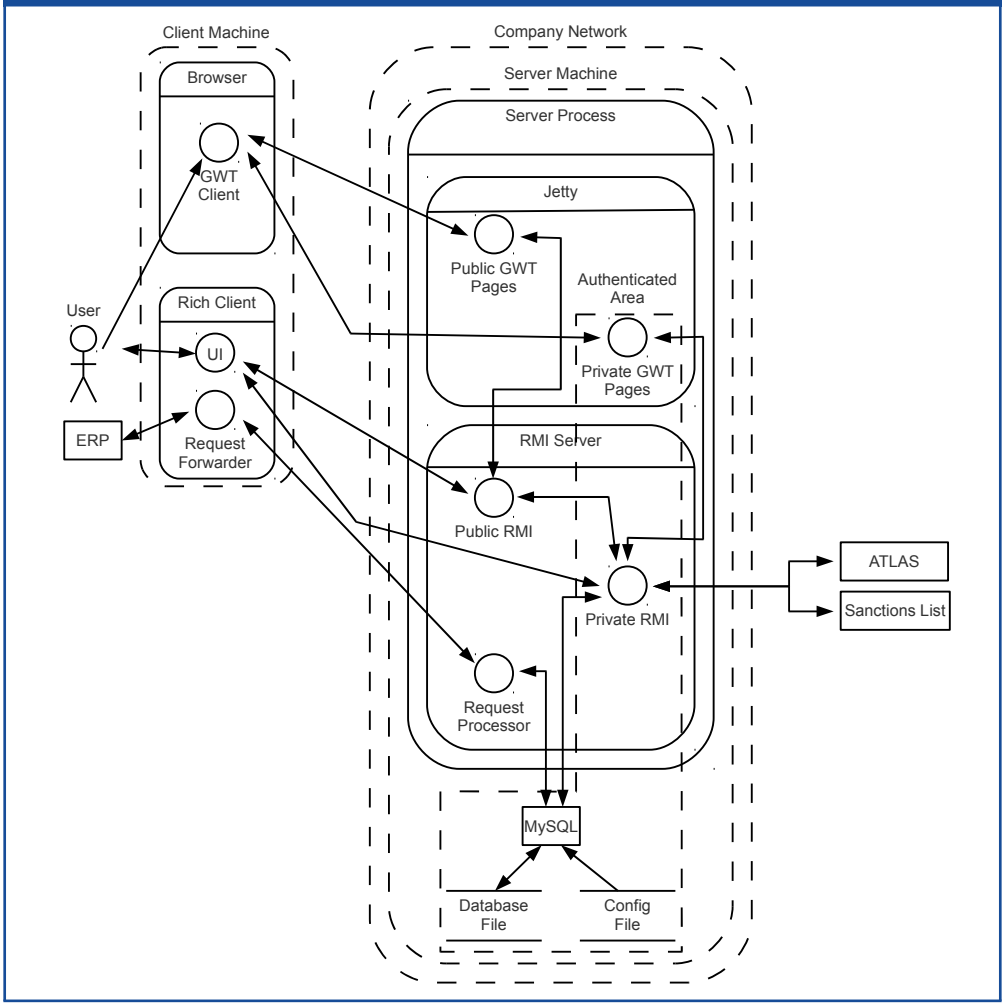
EQ2: What is the impact of the identified security flaws?

EQ3: Are there similarities between the logistics applications?

The first application (Logistics Application A) comes from the logistics domain and is based on a company-specific application framework building on JavaEE technologies. The vendor offers many domain-relevant applications based on this framework and wanted to identify framework-based flaws to improve the security status of their complete product portfolio. The software is

mainly offered on a software-as-a-service (SaaS) base and has several security requirements beyond the functional ones. The application helps customers with customs clearance, and fleet and port management. Figure 12.1 shows a simplified EDFD created during a workshop with three system experts responsible for the architecture and the central development of the framework components. No one of them had deep knowledge in the area of software security at that time.

Figure 12.1. Manually created extended dataflow diagram of Logistics Application A



There are two different clients of the server application. First, there is a *browser-based* client, which is accessible by the end-user. Second, there is a

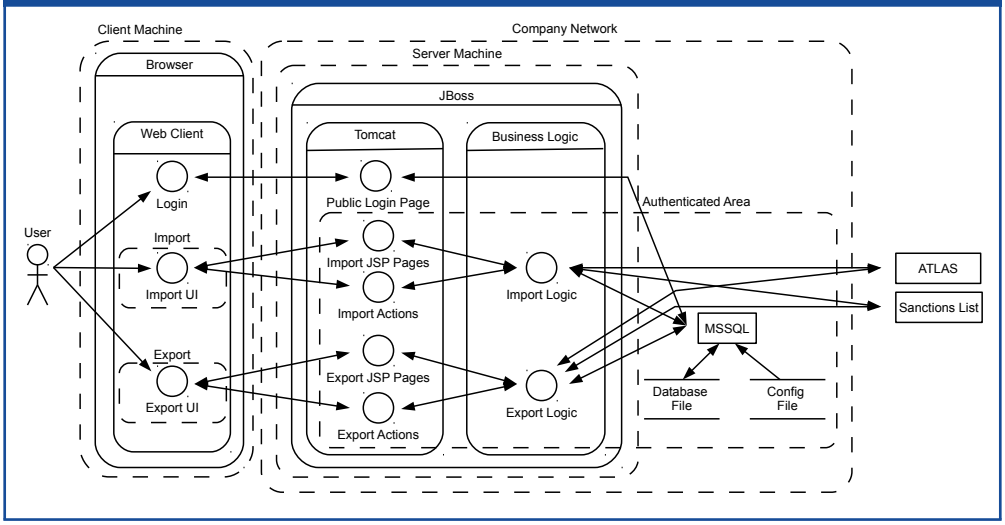
Java-based *rich client* supporting a user interface and an integrated *request forwarder*. The forwarder sends requests it receives from external customer systems, such as *enterprise resource planning software (ERP)*, to the server. The server process and a related SQL-based database are running in the trust area of the *software vendor's network*. The server implements two interfaces for the clients. One of them is a GWT-based web interface, and the second is an RMI-based interface for the rich client. Both interfaces consist of public and private components that require authentication. The clients send credentials to the public interfaces to authenticate the user and receive a session object for identifying the user for subsequent requests in return.

ArchSec identified several possible threats and mitigations for this system that were discussed with the system experts. ArchSec correctly identified that confidential data flow between the clients and the server process. Hence, an attacker can try to capture these data during transmission. Based on the used channel types, it was also found that the channels used TLS for transport encryption and secured the data, thus mitigating the threat. Furthermore, the rule checker found an instance of *CWE-306* and *CWE-288*. The external *ERP system* sends requests to the *Request Forwarder* contained in the *Rich Client*. Then the requests are forwarded to the *Request Processor* on the server-side. The processor, in turn, communicates directly with the *MySQL database*. This communication path does not transport authentication data which indicates that an authentication check is missing. The described communication path has been added to the software since some clients wanted to integrate the logistics application directly into their *ERP* software. Additionally, a possible SQL injection was detected for the private RMI interface. Still, a manual review of the application's source code showed that they used hand-written SQL statements in conjunction with a company-specific API that ensured that SQL injections could not occur. Consequently, this finding is a false positive. Additionally, ArchSec detected that the client implemented authorisation checks rather than the server. Hence, this is an instance of *CWE-602*. The detection process identified even more threats, which were examined in manual reviews, which revealed additional security flaws in this application framework. The consequences and the impact of these findings are high if one considers the sensitivity of the data and systems involved in port logistics. For example, an attacker can circumvent the client-side security checks and access all data and functions available by utilising the request forwarder or modifying the rich client. Since these problems occur in the application framework, each application based on this framework is vulnerable. These vulnerabilities can result in a significant financial loss for the software vendor due to contractual penalties and the economy due to the outstanding delivery of goods.

The second application from the logistics domain (Logistics Application B) is similar to the first one. It helps the manufacturing industry with customs declarations, sanction lists checks, and commissioning. It is implemented based on JavaEE specifications and provides a web-based interface to the customers. The software is distributed and sold on a SaaS basis as well. All customer data is stored in a single database making multi-tenancy an essential topic for the application's security. Due to licensing reasons, the application divides into several components, such as import, export, and sanction lists. In total, there are seven different products that users can buy. The web application's client is a browser and divides it into a public *login page* and a specific area for each product, as mentioned above. These different products are modelled as different trust areas. On the server-side, in the company's network, a JBoss application container runs a Tomcat web container and all product-specific business logic components implementing the view-independent algorithms and the persistence. Tomcat serves different Java Servlets (dynamic web content), a *login page*, *JSP pages*, and *Struts actions* for each product. For persistence purposes, the business logic components interact with a *Microsoft SQL* database running on a different machine. Where necessary, the business logic components communicate with external systems, such as online sanction lists and the German electronic customs interface Atlas. ArchSec identified several existing flaws. First of all, the application contains injection-based vulnerabilities, such as cross-site scripting and SQL injections. The application was vulnerable to these kinds of attacks since the programmers neglected input validation and did not use an SQL abstraction layer such as the Java Persistence API. Furthermore, ArchSec identified a threat based on *CWE-602: Client-Side Enforcement of Server-Side Security*. As for the findings in Logistics Application A, the impact is crucial for the software system's security. It may be easy for attackers to bypass most security measures if they have a valid account for the system. Furthermore, if the communication between client and server is unprotected by appropriate transport encryption, an attacker can steal the credentials of an arbitrary user. Depending on the motivation of an attacker, the vulnerabilities can have different consequences. On the one hand, the attacker can be interested in harming a specific customer and steal sensitive information, such as the list of customers or exported goods. Furthermore, customers can have a financial impact since some taxes are collected automatically on imports or exports in advance. On the other hand, the attacker can compromise the software provider, delete customer data, or disrupt the functionality. Since the contracts contain penalties for delays caused by the software, this finally can lead to financial damage and may have legal consequences.

Regarding the evaluation questions, it is possible to conclude from these case

Figure 12.2. Manually created extended dataflow diagram of Logistics Application B



studies:

EQ1: It is possible to automatically identify threats and security flaws in manually created extended dataflow diagrams. The detected flaws are known from collections such as the *CWE* or *CAPEC*.

EQ2: The impact of the identified flaws is severe. With knowledge of these flaws, an attacker can circumvent most parts of the existing security measures to get or manipulate any of the application's hosted data or access any functionality provided by the applications.

EQ3: The investigated logistics applications both have significant security problems. According to the system experts, the reason for these problems stems from the fact that security was no concern explicitly addressed during software development since the functional requirements were more important at that time. As a result, the security mechanisms were not implemented to make the system more secure but following the existence of functional requirements or since a developer knew it would be a good idea to, for instance, use transport encryption.

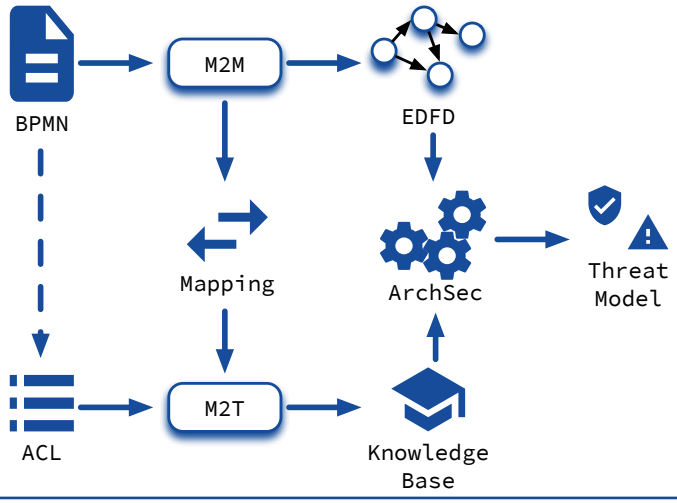
This evaluation used ArchSec to automatically detect architectural security flaws in manually created extended dataflow diagrams. This evaluation

used a knowledge base supporting rules for twelve entries of the *Common Weakness Enumeration*, and five entries of the *Common Attack Pattern Enumeration and Classification* was used. ArchSec detected authentication-related, authorisation-related, cryptography-related security flaws. Besides, it checked for and identified security flaws that do not belong to any of the three categories. Therefore, this case study deals with all research hypotheses in group *H2*.

12.2. Threat Modelling for BPMN

While ArchSec focuses on reverse engineering architectural views from a software's implementation, the situation was different in the research projects PortSec and SecProPort. Here, architectural views in the form of large business process model notation (BPMN) diagrams were present. One research topic, among others, in these projects, was whether these models adhere to an authorisation policy or if there are policy violations. Figure 12.3 shows the outline of our approach to check the authorisation policy.[2]

Figure 12.3. Transformation and checking process



Domain experts created the access control list (ACL) using a custom web tool during the PortSec project. The ACL encodes which subjects of the port community system are allowed to access which information. For example, a subject may be customs or a haulier. Additionally, the passed messages are mapped to

[2]The contents of this section was published at MareSec'21 [33].

assets, which may be sensitive. Therefore, it is possible to automatically extract the subjects and the assets of the ACL from the given BMPN diagrams.

In the first step, a model-to-model transformation converts the given BPMN diagrams into an extended dataflow diagram. The diagrams exist in the form of bizagi Modeler files, which are XML-based model files and therefore computer processable. During this mapping, the transformation encodes different assumptions of the BMMN diagram's structure, such as pools correspond to subjects and swimlanes represent different programs running under the control of the subject. Besides the extended dataflow diagram, this step also produces a mapping of the BPMN diagram elements to the extended dataflow diagram elements.

During the second step, a model-to-text transformation converts the access control list (ACL) to a knowledge base suitable for ArchSec. Therefore, it reads the access control list and the mapping generated during the first step. Next, the transformation generates two knowledge base rules for each asset present in the ACL. The first marks any element processing the asset as a possible threat to the system's security, and the second does the same for every channel transporting the asset to another process. Then, a corresponding mitigation rule is generated for both rules that mark the threat as mitigated. For the first rule, it is sufficient that the process is allowed to process the asset, and for the second rule, the target process of the communication channel needs permission to access the asset. Otherwise, the data is sent to a process that is not allowed to access the asset.

In a final step, the generated extended dataflow diagrams were refined. In this step, additional knowledge of the domain experts was used to make the diagram more precise, e.g., by changing the channel types to proper types, such as HTTPS, SSL connection, or e-mail. Unfortunately, this information is not present in the BPMN diagrams and, therefore, it is not possible to extract it automatically.

Listing 12.1 shows a generated query that identifies channels that transport specific data between two processes. Please note that *data* is the extended dataflow diagram term for *assets*. The query matches the particular asset by its unique identifier in line 2 of the query. Lines 3 to 5 search for a subgraph of the extended dataflow diagram that consists of two elements of type Process and a connecting channel of length one that transports the data matched in line 2. The concluding `WHERE`-clause filters out all channels that start and end at the very same process. As a result, ArchSec marks all channels that transport the specific asset as a threat.

At this point, the mitigation pattern gets crucial since not all transmissions of the asset are a potential threat. Listing 12.2 shows the used mitigation query.

Listing 12.1. Query pattern for channel-based threats

```
MATCH
  (d : Data {"uid" : "«unique-data-id»"}),
  (src: Element {type : subtypeof("Process")})
   -[c : Channel {type : "inter-process communication", data :
↪  contains(d)} * 1]->
  (tgt: Element {type : subtypeof("Process")})
WHERE
  src <> tgt
RETURN
  d AS host, src AS source, tgt AS target
```

Listing 12.2. Mitigation pattern for channel-based threats

```
RETURN
  target.uid IN ["«unique-element-id-1»", "«unique-element-id-2»"]
  AS mitigated
```

The pattern essentially checks if the target's unique identifier is in the list of unique identifiers allowed to access the information. If the unique identifier is found, the query returns true and ArchSec marks the threat as mitigated.

Please note that the given unique identifiers in the shown listings are placeholders for the actual identifiers of the extended dataflow diagram elements. The generated rule and mitigation for processes are similar to the given listings. Therefore, they are omitted here. Besides detecting the threats, ArchSec generates descriptions of the threats. Table 12.2 shows the description template for the shown example.

The shown description template contains different information. First of all, the rule has a name and a description. The description is a template that ArchSec expands. The expressions between an opening and a closing guillemet are evaluated using the actual match in the extended dataflow diagram. Furthermore, the description template assigns a severity and an exploitability category to the threat. The severity of these threats is high, and it is very likely that the threat is exploited since the data are transmitted by the application's design. The threatened protection goals are information disclosure, spoofing, and tampering.

We used the presented approach to check a business process model notation

Table 12.2. Description template of the shown rule

Rule	Violation of authorisation Policy		
Severity	`High`	**Exploitability**	`Likely`

Categories	`Information disclosure`
	`Spoofing`
	`Tampering`		

Description	Process "«source.name»" sends data protected by the authorisation policy to process "«target.name»" using a channel. Since data "«data.name»" is part of the authorisation policy, process "«source.name»" can leak sensitive data. However, the threat might be mitigated if the channel's target is permitted to access the data, which is checked by a corresponding mitigation rule.		

diagram that matched the assessed authorisation policy. We successfully transformed the diagrams and the policy into an extended dataflow diagram and a knowledge base. ArchSec was then successfully able to check the knowledge base within several seconds. The result showed that there were some minor problems, which turned out to be problems with the authorisation policy. The domain experts did not have all aspects in mind while creating the policy. After refining the policy accordingly, there were no policy violations left threatening the system's security.

Nevertheless, it turned out that ArchSec's built-in knowledge base could identify threats based on the refined extended dataflow diagrams. It correctly identified communication channels transmitting sensitive information which did not employ channel or message encryption to protect the transmitted data. In particular, customs sent some export-related information to the port community systems using e-mails. These e-mails are neither encrypted nor digitally signed. Consequently, an attacker can read, discard, or manipulate the sent information.

While Section 10.1 required the security expert to encode the policy with the help of ArchSec rules, this section shows the automation of this idea by generating the rules based on a specified policy. This approach is helpful for users of ArchSec that are not extended dataflow diagram experts or Cypher experts. For them, this approach offers the possibility to define a policy using a UI-based editor.

This section deals with the automatic generation and detection of authorisation-related security flaws, supporting research hypotheses *"H2.2—The proposed approach can automatically detect present authorisation-related security flaws in*

extended dataflow diagrams“. As described in the last paragraph, the described approach helps to define authorisation policies in ArchSec easier and is, thus, a good starting point for even more research.

12.3. Comparison

This chapter presents a comparison of ArchSec's detection mechanism with the approach to detecting architectural security flaws presented by Tuma et al.[3] They introduced a tool for automating the detection of security flaws in manually created dataflow diagrams [160]. Therefore, they enhance dataflow diagrams with possibilities to store information on security patterns, such as *secure pipe* or *authentication*. The security anti-patterns (security flaws) are expressed using Eclipse Viatra patterns [165]. In contrast to ArchSec, distinguishing between security flaw patterns and corresponding mitigation patterns is not possible. Furthermore, their method requires enhancing the diagrams to support new security patterns. Tuma et alùse their approach to automatically search for security flaws in 26 manually created diagrams and compare their results to a manually assessed list of possible security flaws [160].

The following evaluation questions were investigated:

EQ1 Is it possible to automatically convert the dataflow diagrams of the named approaches to one another to create interoperability between these approaches?

EQ2 Is it possible to formulate ArchSec rules that match the description of Tuma et al's security flaw catalogue [160]?

EQ3 Does the automatically converted diagrams lead to similar results when employing the formulated ArchSec rules from **EQ2**?

EQ4 Which effect does the mitigation pattern of ArchSec have on the detection results?

The remaining chapter first presents the evaluation process, then shows the results, and finally the findings concerning the evaluation questions.

It is necessary to use comparable dataflow diagrams and detection rules to compare the results of the detection approaches. Tuma et alpublished an appropriately curated set of 26 manually created dataflow diagrams in 2020 and the findings of their detection approach [160]. The dataflow diagrams are

[3]The content of this chapter is submitted to a peer-reviewed conference but is still under review.

given as instances of an ecore model. They published their findings as a CSV file together with the manually created ground truth. Two security experts developed the ground truth based on their assessment of the 26 diagrams. Tuma et alclassify their findings and calculate precision and recall during their evaluation based on the ground truth.

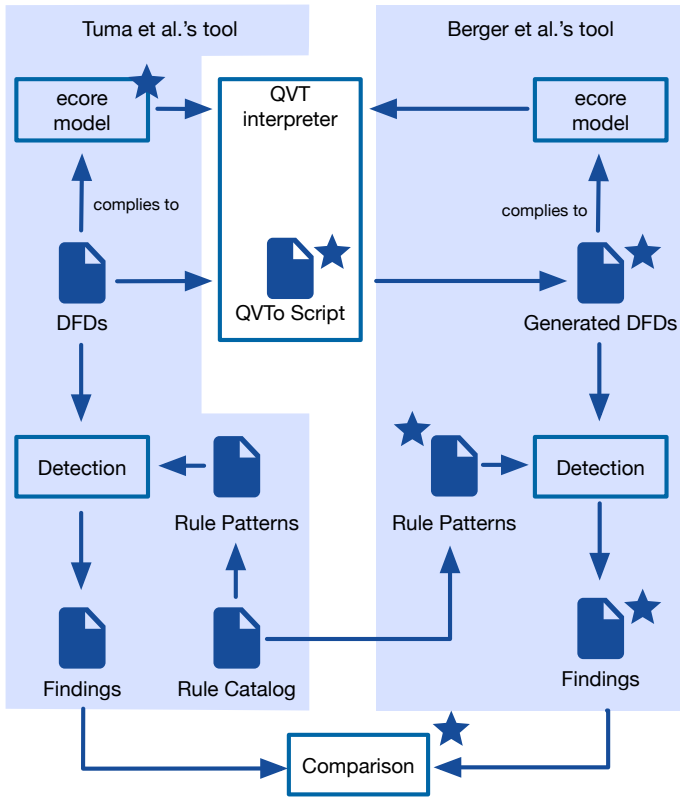
Table 12.3. Supported security flaws in Tuma et al's work.

number	short description	hypothesis
Flaw 2	Authentication Bypass using an alternate path	*H2.1*
Flaw 6	Insufficient Cryptographic keys management	*H2.3*
Flaw 13	Insecure data storage	*H2.3*
Flaw 15	Insecure data exposure	*H2.3*
Flaw 18	Insufficient auditing	*H2.4*

Table 12.3 lists the security flaws Tuma et alinvestigated in their evaluation. The table gives the flaw id, its short description, and states the research hypothesis the flaw relates to. Since they did not inspect authorisation-related security flaws, no rules are assigned to *"H2.2—The proposed approach can automatically detect present authorisation-related security flaws in extended dataflow diagrams"*. Appendix C gives a full description of the security flaws.

Figure 12.4 shows the evaluation's outline where all extensions to ArchSec are marked using a star. The left-hand side depicts the results of Tuma et al. They used 26 data flow diagrams that complied with their ecore model. These diagrams are the input of the detection process, which applies rule patterns to them and creates a list of findings. The authors created the rule patterns based on their presented textual rule catalogue. The same dataflow diagrams were analysed with ArchSec to compare the detection approaches. Therefore, an import of Tuma's dataflow diagrams was implemented. For this purpose, it was necessary to re-create the ecore model of Tuma et al's diagrams since it is not publicly available. Based on this ecore model, a QVTo-based model-to-model transformation [120] was specified to convert between these two models. To check the same security flaw patterns, it was then necessary to create a rule catalogue implementing the same rules as the ones used by Tuma et alin their evaluation. ArchSec then searched for instances of the rules in the translated dataflow diagrams. The results are then compared to the manual analyses results (ground truth) provided by Tuma et al[160] as well as the results of their automated detection process. For the comparison to the manual analyses results, we use the standard measures *precision* ($precision = \frac{TP}{TP+FP}$) and *recall*

Figure 12.4. Evaluation design



($recall = \frac{TP}{TP+FN}$), where TP means *true positive*, FP means *false positive*, and FN means *false negative*.

The created transformation can successfully convert all 26 dataflow diagrams of Tuma's data set. The QVT-model-to-model transformation resulted in EDFDs compliant with ArchSec's model, and a manual inspection showed that they convey identical information although they differ in their visual depiction. For example, Tuma et aldepict a security pattern through a node with edges to the protected parts of the basic dataflow diagram, whereas ArchSec models such security patterns via attributes on edges and nodes. The conveyed information, nonetheless, is identical.

ArchSec's detection algorithm results with the newly defined queries for security flaws 2, 6, 13, 15, and 18 of Tuma et al's catalogue are compared to the supplied *ground truth*. Tables 12.4, 12.5 and 12.6 show the received precision, recall, and f-measure differentiated according to the supplied system

and research hypothesis investigated. Figure 12.5 shows the same data (but not aggregated to the research hypotheses) in a graphical form. Every data point visualises precision (x-axis) and recall (y-axis) for a given system and security flaw (shapes representing different flaws). The curves represent the area for obtaining a specific f-measure, i.e. data points on the outer side of this curve represent a higher f-measure.

Appendix D lists the detailed findings behind these summarised results.

Table 12.4. Summary of the results for research hypothesis *H2.1*.

		BeSocial	*DriveSafe*	*PhotoFriends*	*SmartTex*	*Total*
H2.1						
	ArchSec					
	precision	*0.26*	*0.20*	*0.40*	*0.58*	*0.39*
	recall	*0.40*	*0.22*	*0.46*	*0.52*	*0.43*
	f-measure	*0.32*	*0.21*	*0.43*	*0.55*	*0.41*
	Tuma					
	precision	*0.21*	*0.13*	*0.47*	*0.48*	*0.35*
	recall	*0.40*	*0.22*	*0.64*	*0.57*	*0.50*
	f-measure	*0.28*	*0.16*	*0.55*	*0.52*	*0.41*

Regarding research hypothesis *H2.1*, Table 12.4 shows that both approaches perform similarly. While the precision is, on average, slightly better for ArchSec, Tuma et al's approach has a slightly better recall, resulting in an f-measure of 0.41 for both approaches. For research hypothesis *H2.3*, Table 12.5 shows that ArchSec's results are, on average, slightly but not significantly better than the findings by Tuma et al. On average, precision, recall, and, accordingly, f-measure are better for ArchSec. Lastly, Table 12.6 shows that the results for research hypothesis *H2.4* give a more diverse picture. Here, ArchSec produces noticeably better results than Tuma et al's approach. While the precision is 0.12 percentage points better, the recall is 0.55 percentage points better.

Additionally, the nature of the received false positives was investigated as the precision is, on average, worse than the recall for all research hypotheses. To

Table 12.5. Summary of the results for research hypothesis *H2.3*.

		BeSocial	*DriveSafe*	*PhotoFriends*	*SmartTex*	*Total*
H2.3						
	ArchSec					
	precision	*0.26*	*0.65*	*0.65*	*0.61*	*0.64*
	recall	*0.95*	*0.90*	*0.80*	*0.86*	*0.87*
	F1-measure	*0.77*	*0.75*	*0.69*	*0.74*	*0.74*
	Tuma					
	precision	*0.62*	*0.64*	*0.66*	*0.49*	*0.60*
	recall	*0.93*	*0.89*	*0.77*	*0.82*	*0.85*
	F1-measure	*0.75*	*0.74*	*0.71*	*0.61*	*0.70*

that end, a comparison of Tuma's *false positives* and ArchSec's took place to get more insight into these findings. Therefore, the ratio of unique false positives per approach was calculated. The following formulas calculate these ratios:

$$ratio_{\text{ArchSec}} = \frac{|FP_{\text{ArchSec}} \setminus FP_{Tuma}|}{|FP_{\text{ArchSec}}|}$$
$$ratio_{Tuma} = \frac{|FP_{Tuma} \setminus FP_{\text{ArchSec}}|}{|FP_{Tuma}|}$$

Figure 12.6 shows the ratio of detection-procedure-specific *false positives*, i.e., the ones found by ArchSec but not Tuma and vice versa, to all detected *FP*s by that method. A high ratio of *false positives* that are specific to a given approach can be interpreted as an imprecise implementation of the security flaw compared to the rule interpretation of the security experts creating the *ground truth*.

The results show that some of the rules, especially rule 6, result in nearly distinct sets of *false positives* while other rules, such as rule 13, have very similar

Table 12.6. Summary of the results for research hypothesis *H2.4*.

		BeSocial	*DriveSafe*	*PhotoFriends*	*SmartTex*	*Total*
H2.4						
	ArchSec					
	precision	*0.23*	*0.45*	*0.44*	*0.50*	*0.39*
	recall	*0.70*	*0.83*	*1.00*	*0.92*	*0.87*
	F1-measure	*0.34*	*0.59*	*0.92*	*0.65*	*0.54*
	Tuma					
	precision	*0.25*	*0.00*	*0.57*	*0.29*	*0.27*
	recall	*0.25*	*0.00*	*0.80*	*0.33*	*0.32*
	F1-measure	*0.25*	*0.00*	*0.67*	*0.31*	*0.29*

result sets. These results indicate that the natural language rule descroptions found in the pattern catalogue that were used are different in their preciseness. While rule 13 seems to be very precise, rule 6 leaves much space for interpretation, resulting in diverse findings.

ArchSec defines mitigation patterns that allow them to retract identified security flaws when they are mitigated. For example, an insecure data exposure (Flaw 15) can be mitigated through a secure channel. For the results shown above, the comparison only considered flaws that were not mitigated, i.e. actual flaws (in terms of this detection). However, when analysing the mitigated flaws of ArchSec's detection, we see that all but 2 of 93 findings are indeed *false positives*, that is, they were correctly identified as mitigated. Furthermore, 61 of those 93 mitigated flaws are found by Tuma's detection as well, but—as they are not mitigated in their approach—they are identified as *false positives* in their evaluation.

Considering **EQ1**, the obtained identical information shows for 26 different dataflow diagrams that it is indeed possible to automatically convert dataflow diagrams between these approaches yielding interoperability.

EQ2 and **EQ3** strongly influence one another. If the formulation of graph queries is not possible, the search for security flaws with ArchSec will not

Figure 12.5. Comparison of precision and recall for Tuma et al's detection scheme and ArchSec's detetion scheme grouped by System and Security Flaw

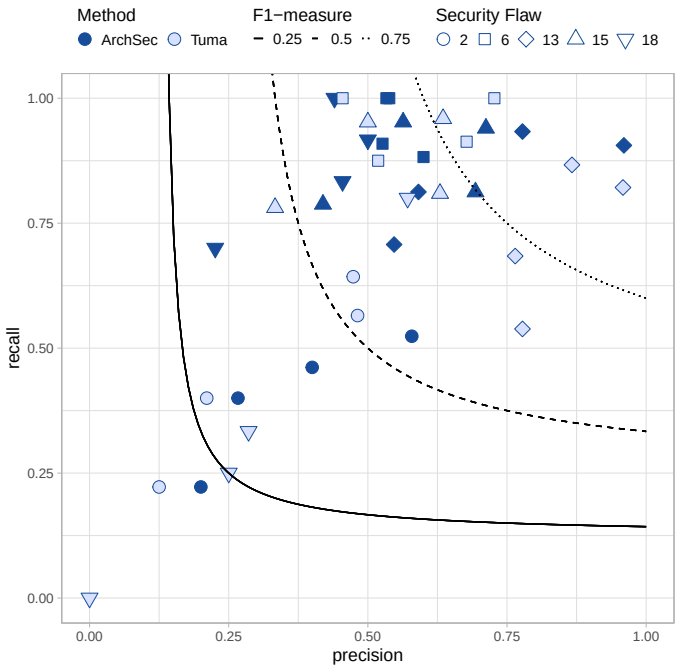
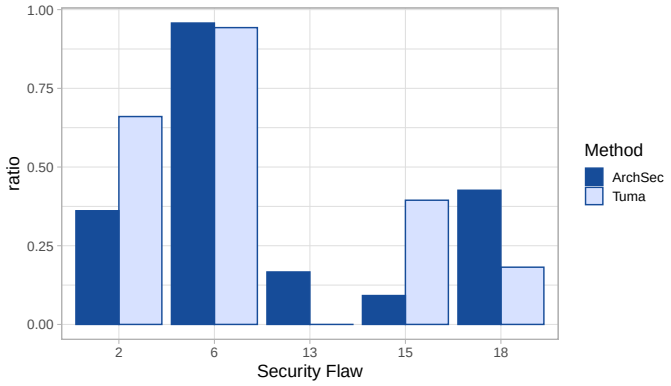


Figure 12.6. Ratio of method-specific false positives to all false positives found by that method



be successful. Therefore, the results of the actual detection also give hints to the appropriateness of the formulated queries. *Security Flaws 13: Insecure data storage* and *15: Insecure data exposure* have a good recall and reasonable precision. Additionally, they are comparable to Tuma's results. Hence, for these flaws, both questions can be answered positively. For *Security Flaw 6: Insufficient key management* the precision is weaker (compared to the other flaws) but similar to Tuma's. On the other hand, ArchSec finds almost completely different *false positives* than Tuma (see Figure 12.6). This might imply an unclear definition in the catalogue rather than the graph query as both automated approaches are comparably worse in numbers but differ in regards to actual results.

Security Flaw 18's precision is worse than the three already mentioned flaws, but its recall is comparable (in terms of flaws). Additionally, its recall outperforms that of Tuma et al greatly (see Figure 12.5). More than half of the false positives are found by both automated approaches, which allows the suspicion that this might be a formalisation issue compared to the manual inspection for the ground truth. All in all, we can state that for 4 out of 5 security flaws, the formulation indeed was possible, and the obtained results are comparable and slightly better than those produced by Tuma et al [160]. Including mitigations into an automatic security flaw detection proves to be an important design decision. It reduced the number of *false positives* significantly (especially in comparison with the results of Tuma) and produced only two *false negatives*. Please note, however, that for both *false negatives* the thesis' author does not agree with the *ground truth* after manual inspection.

There are several threats to the validity of the presented results. First, the comparison of the detection approaches depends on the validity of the used ground truth. As already mentioned, the thesis' author does not agree with all entries in the ground truth. Moreover, in his opinion, some entries are missing. Nevertheless, the small number of these elements will not fundamentally change the evaluation results. Another threat to the validity of the evaluation is the way the results of the detections are matched to the ground truth. The specified rules are intended for human beings, allowing different interpretations. The results are automatically compared based on the rule id, the concerned element, and an optional dataflow. While the ground truth and ArchSec's results specify additional comments explaining the finding, the results of Tuma's evaluation contained no further comments. In some cases, the set of compared elements were not distinct, resulting in a fuzzy matching of some results. A last threat to the validity is the preciseness of the security flaw detection guideline coded into automatically detectable rules. If two attempts to automate the rule result in totally different patterns, it might indicate that human beings using the

catalogue in a manual review have similar problems.

12.4. Chapter Summary

This chapter dealt with detecting security flaws in manually crafted extended dataflow diagrams. While Section 12.1 deals with all categories of security flaws—supporting all research hypotheses from the second group—Section 12.2 focuses on authorisation-related security flaws—supporting research hypothesis *H2.2*. Finally, the last section of this chapter contributes to research hypotheses *H2.1*, *H2.3*, and *H2.4*. Additionally, it compares ArchSec's detection approach with another approach of automatically detecting security flaws. Both approaches performed similarly, while ArchSec's way of modelling dataflow diagrams and detecting security flaws seems to be more flexible.

Part IV

Finale

Chapter 13

Related Work

As described in the Prelude, three broad research topics relate to this thesis' topic (c.f. Figure 1.3). First, software security deals with threat modelling and possibilities to mitigate security threats. Second, software architecture focuses on planning and describing software systems. Third, static analysis works on identifying bugs or extracting higher-level information from software systems. Accordingly, plenty of research is related to at least one of the thesis' research topics. To identify relevant related research, Section 13.1 describes the process of the conducted literature search, while the remainder of this chapter summarises the related research found.

13.1. Literature Search

The literature search aims to identify research that deals with topics similar to the thesis' topic and uses the snowballing process [168]. The snowballing is a graph traversal algorithm that starts with an initial working set of publications from a simple literature search. Then, for each paper of the set, previously defined inclusion and exclusion criteria are checked. If a paper is not excluded, it becomes part of the final set of relevant publications. When added to the final set of relevant publications, all cited papers are added to the working set if they are not part of the final set already. Additionally, search engines are used to identify publications referencing the added paper. These are added to the working set as well.

To identify the initial set of interesting publications, the literature search has a twofold input. First, it uses different search engines for academic literature, and second, it takes the thesis author's publications into account. The search engines receive a list of relevant search terms to conduct their search. Table 13.1 lists all used search terms and respective synonyms. The search aimed to identify research that deals with threat modelling and static analysis. The used search engines influenced the chosen synonyms since one engine expected exact string matches for keywords. Therefore, the synonyms are noted

in British and American English since publications use both language variants. Additionally, it contains hyphenated and non-hyphenated versions of the search terms. Supplementary, the synonyms architectural risk analysis, security assessment and risk analysis in combination with software architecture are used. The synonyms program analysis and reverse engineering complete the search terms.

Table 13.1. List of all Used Search Terms

search term	synonym
threat model	threat modell
	threat-model
	threat-modell
	threat-modeling
	threat-modelling
	security assessment
	security-assessment
	architectural risk analysis
	architectural-risk-analysis
	risk analysis AND software architecture
	risk-analysis AND software-architecture
static analysis	static-analysis
	program analysis
	program-analysis
	reverse engineering
	reverse-engineering

IEEExplore[1] and ACM Digital[2] were the engines of choice to search since they support the possibility to combine search terms using logical operators. The search engines Google Scholar and CiteSeerX do not list such options and therefore were not used during the first step of identifying relevant work. IEEExplore returned a list of 35 publications, while ACM Digital Library found 807 publications. The related research sections from former publications contained 79 publications, and another 58 publications cite these publications. Merging these sets resulted in 960 interesting publications for the snowballing process.

The exclusion criteria serve the goal of removing publications that are not related to the topic of interest. An identified publication is seen as unrelated if it meets one of the following criteria:

[1] https://ieeexplore.ieee.org
[2] https://dl.acm.org

Authorship A publication is excluded if the thesis' author is one of the publication's authors (except for an initial forward snowballing).

Non-Software A publication is excluded if it is not related to software systems.

Subject-Miss A publication is excluded if threat modelling is only mentioned in the introduction or the related research.

Books Books explaining the Threat Modelling process are excluded.

Research Proposals Publications reporting on planned research are excluded.

Additionally, publications that discuss the advantages and disadvantages of the threat modelling process are included to give additional ideas for research.

Applying the exclusion criteria to the initially found set of papers, resulted in 29 papers of interest. The snowballing process then identified additional twelve papers of interest in three rounds, resulting in a total of 41 papers that are part of the this chapter.

13.2. Security in General

In 2009, Rehman and Mustafa reported on design-level security vulnerabilities and their roots. According to the authors, there was a lack of research on security integration in the design phase, although the importance had been claimed in many different locations. They reported on various topics in the area of secure design and on existing research. Furthermore, they identified research subjects that needed further investigation. Topics of interest, among others, are tools to detect existing vulnerabilities in design, tools to model security policies in design, threat modelling with more semantic and context information, and security patterns at the architectural level [130]. Thus, the publication underlines that the thesis' research topic is a relevant one.

Pieczul, Foley, and Zurko described the fact that the type of people who are involved in developing software systems changed over the years. Therefore, the security design of applications and software libraries changed as well. They claimed that software developers had been highly-skilled technology professionals, but it has shifted towards a continuum from end-users over administrators to still highly specialised programmers. One problem the authors identified is the fact that most of these developers are not familiar with security concerns. Due to their ignorance, they might introduce security vulnerabilities while using third-party software and libraries because they are simply not aware of the

security implications of their programming. Furthermore, the authors claimed that there had been a shift from user-centric security towards developer-centred security and that some of the user-centric security paradigms might not be suitable for developer-centric security. One approach they mentioned that is suitable for developer-centric security is Microsoft's threat modelling and the employed STRIDE approach [125].

13.3. Securing the Software Development Process

Tøndel, Cruzes, and Jaatun discussed security in the context of agile development. They stated that there is no perfect security, and therefore security-related risk-management is a necessary task. They point out that a robust security design is necessary to build a secure system but that it is not sufficient for creating a perfectly secure system. Using surveys, they tried to find out how agile software development teams deal with security and how it can be integrated into agile software development. They observed the usage of different security practices, such as threat modelling, static analysis tools, self-management for security, and security requirements work [156].

Dejan Baca and Bengt Carlsson stated that there are concerns that the higher development pace and lack of documentation in agile software development are creating less secure software. One reason for this assumption is the fact that there cannot be a complete picture of the product since not all requirements are known in advance. This lack of completeness might make it impossible, e.g., to apply some architectural security concepts. To validate this assumption, they analysed secure software development processes, such as Microsoft's SDL, and cigital's touchpoints, regarding the used means and then interviewed developers from a large telecommunication manufacturer whether these means were employed in their agile projects. The interviews showed that the identified security means scale badly to agile development since they are too expensive or not beneficial enough—especially the means from the design and test phase scale badly [21].

Rindell, Hyrynsalmi, and Leppänen investigated the question: "How can the agile practices be combined with software security engineering activities?". According to the authors, agile software development processes are lacking the ability to produce secure software since they are value-driven processes. They extracted the different security activities from Microsoft's SDL, the Common Criteria, and OWASP SAMM and mapped them to the various classic software development phases. Among these activities, there are *Use Threat Modelling*, *Security Architecture*, and *Design Review*. Afterwards, the authors mapped

them to the different parts of agile development, such as the product backlog, daily meetings, and test-driven development [132].

Maier, Ma, and Bloem proposed integration of security activities into the Scrum process for developing secure web applications. They adapted the security engineering activities from the ISO standard *System Security Engineering - Capability Maturity Model* to Scrum. They proposed an agile risk analysis method based on *OWASP's Application Threat Modelling*. The authors then evaluated their approach with ten Scrum developers. They used a questionnaire to ask for the additional time necessary, the costs, and the agility of the propose changes. The authors used the median of the answers to conclude that their approach is medium agile and medium cost-effective [113].

13.4. Modelling Architectures

Cherubini et al. have examined how and why developers use which type of diagrams with the help of several interviews and a structured survey [51]. Their research found that manual diagrams are used when developers are communicating face-to-face. According to the authors, this stems from the problem that none of the available tools was sufficient for the developers to write down their mental model of the software. In particular, the changing degree of abstraction during the discussions is problematic for existing tool-based approaches. The diagrams to describe different aspects of the software shown in the publication were of box-and-arrow nature. According to the participants, the favourite way of visualising software are box-and-arrow drawings on whiteboards or a piece of paper. Extended dataflow diagrams use this fundamental idea and allow the modelling of various levels of abstraction due to their hierarchical and extensible schema structure.

Yskout et al. took a look at the development of threat modelling and identified challenges for using current threat modelling ideas in practice. The first task they identified is the development of a reference model for threat modelling that allows the exchange of threat modelling artefacts. Second, they see the necessity to research means to capture the development of a threat model to document the evolution of a threat model. Third, the authors suggest developing a knowledge base of threat modelling knowledge. Fourth, Yskout et al. identified the need for metrics that capture the threat model's quality. Last, they suggest integrating runtime information into a threat model to make it dynamic [161].

13.5. Security Flaw Detection at the Architectural Level

Jürjens et al. published several papers on *UMLsec* between 2001 and 2012 [40, 90–93, 136]. *UMLsec* is a UML profile aiming at specifying and checking security requirements. Within their publications, the authors focus on different aspects of cryptography and cryptographic protocols. In addition, they took a look at authentication. The security-relevant aspects of a software system are added to UML models using UML stereotypes, tagged values, and constraints. Their approach adds the information to activity diagrams, state charts, sequence diagrams, static structure diagrams, and deployment diagrams. Jürjens et al. use the *Spin* model checker to verify the security properties, such as *secure link*, *secrecy*, or *no down-flow*. A component with the stereotype *secure link* has to make sure that all outgoing connections are secure. A dependency of the stereotype *secrecy* has to provide secrecy for the sent data. If a subsystem has the stereotype of *no down-flow*, it has to make sure that no message is sent from a secret to a non-secret subsystem. In contrast to ArchSec, *UMLsec* does not extract the checked model from the implementation. Instead, it only checks a manually created model. The used UML diagrams allow to describe more aspects of a software system due to the available diagram types, but the information has to be added over several different diagrams in order to be checked. Additionally, *UMLsec* is a constructive approach. It checks the annotated security requirements. Therefore, it is not able to identify existing vulnerabilities if the software architect did not annotate the UML diagrams correctly.

Basin et al. introduced *SecureUML*, a forward engineering approach, for specifying RBAC policies for JavaEE systems using UML diagrams [26, 27, 112]. The required permissions for using a method or a class are annotated using association classes and additional OCL-based constraints. Furthermore, the authors present two possibilities to use the added policy information. First, they specify scenarios (in the form of UML object diagrams) and check their correctness by evaluating the OCL constraints. Second, they generate deployment descriptors for the JavaEE container. These deployment descriptors encode the authorisation policy, and thus, the container enforces the policy defined in the UML diagrams. This approach has the advantage that there is no gap between the planned and the implemented authorisation policy. *SecureUML* shows that it is possible to integrate authorisation information into the software architecture. Furthermore, it successfully used the model to check authorisation automatically and was, therefore, a first evidence for the feasibility of automatic security flaw detection.

In 2007, Abi-Antoun et al. describe an approach to facilitate DFDs for identi-

fying security flaws [9]. The authors describe DFD's as a runtime view following the Component-and-Connector view type, according to Clement et al.'s definition (c.f. [79] for more details). The proposed approach first extracts an as-implemented DFD from a software's binary and compares it to a given as-designed DFD. An extended version of Murphy et al.'s reflexion model [116] conducts this comparison and yields a list of correct, missing, and unplanned dependencies between the architectural entities (DFD nodes in this case). Furthermore, they verbally describe a set of rules that leads to possible architectural-level threats. The paper gives a textual representation of the DFD model they use. However, it does not explain how the DFDs are extracted and how the rules are applied to them. Abi-Antoun and Aldrich introduced Scholia, an approach to statically extract hierarchical runtime architectures of Java code using annotations, in 2009. Based on the annotations, they gather object-ownership graphs and analyse the conformance to a given architecture. In 2010, Abi-Antoun and Barnes applied the Scholia approach to the security domain and created the Secoria approach. They extended Scholia to extract hierarchical object graphs and map the result to dataflow diagrams. The extracted DFDs are then checked using well-formedness, information flow, and general constraints. The well-formedness constraints validate the correct extraction of the DFDs, such as if there are dataflows between data stores without a process. The information flow constraints search is limited to the detection of access control list violations. The general constraints let the user define application-specific first-order logic predicates to detect unwanted communication in the system. The authors successfully evaluate Secoria using a Java application consisting of 3000 lines of code [8]. Vanciu and Abi-Antoun formalised their analysis which they presented in 2012 and showed their analysis' soundness. The presented extension uses a domain-specific language on top of Java to add ownership information to the code. They apply their approach to CrypoDB, a Java-based database implementation that consists of 3000 lines of code and compared the extracted object-ownership graph with a manually created dataflow diagram. The comparison showed that their approach can extract a dataflow diagram similar to the manually created one [164]. In 2014, Khalaj, Vanciu, and Abi-Antoun used Secoria to investigate whether there is value in reasoning about security at the architectural level. They compared Secoria to a specialised dataflow analysis called FlowDroid [20] and compared the identified findings. They positively answer their research question, since—based on the chosen benchmark—Secoria's precision, recall and f-measures are better. While the work conducted by Abi-Antoun et al. uses static analysis to extract architectural information and uses dataflow diagrams as well, their approach does not support an extensible way of describing security flaw patterns. Addi-

tionally, in contrast to ArchSec, their approach requires developers to add annotation-based information or to switch to a domain-specific language to guide their analysis.

Antonino et al. presented an early research prototype for indicator-based architecture-level security evaluation for service-oriented architectures, called SiSOA, in 2010 [19, 89]. Their approach is made up of three steps. First, according to the publication, the extraction phase uses reverse-engineering techniques to create a system model that comprises static class-level information from the source code and configuration information. Second, the identification phase employs a knowledge base to tag the system model. The tag rules can either add abstract information about the program, such as identified SOA components, or security information. Finally, a severity and a credibility value are calculated and assigned to the added tag for each tag. These metrics quantify the quality and correctness of the placed tags. The calculated tags are then used to check the conformance to manually defined security goals, such as message integrity. The authors demonstrate their approach with an example but do not provide an evaluation using a real-world application. The SiSOA idea is an exciting approach to architectural security flaws but lacks thorough static analyses since the presented static information are all at the class level. Since the methods are not taken into account, the indicators are a mere heuristic approach to detect security means. Consequently, ArchSec is more powerful due to the employed analyses.

Alshammari, Fidge, and Corney defined hierarchical security metrics to assess the security of an object-oriented software system [15]. They argued that few work had been done on measuring the overall security of a given program concerning information flow. The root of their metrics is the manual classification of attributes into classified and non-classified ones. They then used Java bytecode analysis to extract implementation-level metrics. They defined metrics on four hierarchical levels that build on one another. On the implementation level, they used 25 metrics for measuring the potential flow of classified data. The next level consisted of seven metrics describing the readability and writability of classified data. On the third level, the aforementioned metrics are aggregated to seven security design principle metrics: *grant least privilege*, *reduce the attack surface*, *secure the weakest link*, *fail-safe defaults*, *least common mechanism*, *isolation*, and *economy of mechanism*. Finally, the highest metric level combines all metrics to a total security index. In 2012, Alshammari, Fidge, and Corney investigated the impact of refactorings in object-oriented software on the software's security [16]. To determine the effect of refactorings, they used their previously defined dataflow metrics, measured them before and after the refactoring, and compared the results. In particular, they applied the re-

factorings *Add Parameter*, *Remove Parameter*, *Decompose Conditional*, *Replace Delegation with Inheritance*, *Replace Inheritance with Delegation*, *Replace Fields with Subclasses*, *Replace Subclasses with Fields*, *Replace Array with Object*, *Replace Object with Array*, *Replace Temp with Query*, *Replace Query with Temp*, *Remove Setting Method*, *Parametrise Methods*, *Replace Parameter with Explicit Methods*, *Replace Method with Object*, and *Replace Object with Method*. Based on their case study, the authors concluded that only the refactorings *Parametrise Method* and *Remove Setting Method* harm the overall application's security. Unlike ArchSec, the presented approach tries to quantify the security of a software system and does not identify concrete security flaws.

In 2012, Almorsy, Grundy, and Ibrahim introduce an approach to detect implementation- and architectural-level security flaws based on formalised signatures. A vulnerability signature is expressed using an adapted version of OCL and consists of a set of invariants. The approach then first extracts an intermediate representation. Then the invariants are applied to the intermediate representation, and the described vulnerability exists if the invariants hold. The authors list signatures for *SQL injection*, *cross-site scripting*, *improper authorisation*, *cross-site request forgery*, *information exposure*, *URL redirection* and *improper authentication*. These are all vulnerabilities that may occur in dynamic web applications [13]. In 2013, Almorsy, Grundy, and Ibrahim extended their approach to support security-relevant metrics and more architecture-level security flaws. They added signatures for the *attack surface metric*, *compartmentalisation metric*, *least privilege metric*, *fail securely metric*, *defence-in-depth*, and *isolation metric*. The newly added security flaws are *man in the middle*, *denial of service*, and *data tampering*, [14]. In contrast to ArchSec, the presented approach extracts UML diagrams for .NET based software systems and uses OCL for describing security rules.

Välja et al. presented an ontology-based approach to threat modelling [161] in 2020. The work focuses on creating threat models based on software products, operating systems, other types of applications, and dataflows. Their approach works on dynamic network information and does not analyse software systems. Consequently, this idea complements the ideas presented in ArchSec.

13.6. Threat Modelling

Sion et al. proposed a more detailed risk estimation for threat modelling in 2018. According to the authors, it is necessary to estimate four different aspects for each threat. The first aspect is the strength of the countermeasure. Second, the attacker's strength is estimated. This estimation reflects her ability

to attack the system. Third, the contact frequency of the attacker with the system is estimated. Last, the probability is estimated how likely it is that an attacker—coming into contact with the system—will attack the system. This improved risk estimation is used to rank the identified threats [146]. In 2019, Sion et al. analysed 19 security flaws to identify necessary information for their automated detection. On the basis of the results, they presented extensions to dataflow diagrams for expressing these flaws. According to the authors, the following extensions are needed: First, information on data and its transformations. Second, they identified the necessity for dataflow diagrams to store information on security countermeasures [145]. In the very same year, Sion et al. published an arcticle focusing on threat modelling for privacy. They presented an architectural viewpoint for data protection that they created according to the requirements imposed by the GDPR. They claim that their view is complementary to the existing dataflow diagrams and that they keep it consistent with the help of correspondence rules. Both views are aligned, and the analysis activities for creating these reinforce each other [144].

Katja Tuma et al. published a structured literature review on threat analysis of software systems [157]. They report on 62 papers related to threat analysis in general which makes the literature review broader than the one given in this thesis. The authors categorised the identified papers according to different aspects, where one of them is the category precision. It has the categorical values *none*, *based on examples*, *based on templates*, *semi-automated*, and *very precise*. The papers classified as semi-automated and very precise were of interest for the snowballing since they might be related to ArchSec. The literature review also reports on one of the publications that are part of this thesis:

> *However, if the planned resource investment is "large", the organisation is likely prepared for improving an existing technique to obtain an "in-house" adapted version. We also consider academic researchers looking for a starting point in their research to be prepared for a "large" investment of resources. These techniques should be systematic by construction (e.g. formal) but most importantly show potential for improvement (e.g. technology improvement). In our opinion, two such techniques stand out. First, the work of Berger et al. [38] presents an interesting semi-automated technique for extracting threats from graphs based on rules matching certain CAPEC and CWE entries. The authors argue that the existing notation for DFDs needs to be extended with more security semantics. To this aim, Berger et al. extend the notation by annotating flows with assets, security objectives and type of communication (e.g. manual*

> *input). A more formal definition of security semantics might assure the quality of outcomes explicitly, which is a promising research direction. Further, querying graphs could be implemented using a different set of technologies. Therefore we believe that their approach is with some effort adaptable to the needs of the organisation.*

In 2017, Tuma et al. proposed an extension to dataflow diagrams. This extension allows end-to-end asset flows and the possibility to add security assumptions. They demonstrate their approach by modelling a cyber-physical system. Furthermore, they deal with threat explosion. According to the authors, threat explosion is the fact that the number of irrelevant threats is very high during threat modelling. They deal with this problem by reducing the number of elements within the dataflow diagram. Therefore, they bundle dataflows and fold processes [159]. Two years later, Tuma et al. presented two empirical studies on threat modelling using a guideline. In total, six persons participated in that study, and their task was to identify threats in a given dataflow diagram. It was not part of the task to create a dataflow diagram. The precision of the guided threat identification was 0.926, and the recall was 0.504. Nevertheless, the small number of participants does not allow to generalise the results [158]. Tuma, Sion et al. extended the guideline idea in 2020 and now support an automated detection of some of the threats. Nevertheless, they state that it is impossible to automate the detection of all threats since the guidelines are fuzzy, and thus, the instructions are not complete. They evaluated their automation using an empirical evaluation to determine precision and recall of their approach [160]. Similar to ArchSec they use query patterns—in their case based on VIATRA—to identify threats. In contrast to ArchSec they do not analyse source code to create dataflow diagrams but work on manually created ones.

13.7. Case Studies

Karppinen et al. published a case study on using static and dynamic analyses for detecting security vulnerabilities. Their case study used the Software Architecture Visualization and Evaluation (SAVE) [149] tool and compared static and dynamic dependencies of an automated air traffic control system, which was designed to aid air traffic controllers in detecting and resolving conflicts between aircrafts. The comparison allowed them to identify a backdoor that was added to the system by the developers. While the evaluation was successful and showed the possibility of detecting architecture-level security flaws, it is questionable if the approach works for problems unknown in advance [95].

Almohri et al. showcase a threat model for medical cyber-physical systems. A medical cyber-physical system has different safety and security requirements depending on its purpose. Such as system can monitor sensitive patient data or assist during patient treatment. For creating the threat model, the authors use an exemplary high-level architecture of medical cyber-physical systems. Then, based on the roles of the stakeholders, Almohri et al. conduct the threat modelling, identify possible threats, and derive necessary security measures [12].

Dhillon reports on his experiences with real-world systems at EMC corporation. Dhillon presents annotated dataflow diagrams that host additional information on existing security measures. In addition, he gives an idea of a manual threat library maintained at EMC, which contains threats such as *"Unauthenticated access through user interface"*, *"Authorisation bypass"*, or *"Tampering in transit"*. Dhillon underlines the usefulness of the threat library for non-security-aware developers for finding security flaws [63].

Dhillon and Mishra reported on the dynamic verification of manually created threat models. They stated that checking a manually created threat model for a given software system against its implementation took an expert penetration tester six to ten weeks. It was necessary to customise off-the-shelf tools to that end. The checked vulnerabilities ranged from classical cross-site scripting and cross-site request forgery to authorisation, authentication, and weak cryptographic issues [62].

13.8. Chapter Summary

This chapter presented related work based on a snowballing-based literature search. The snowballing was conducted to identify known and unknown research on automating threat modelling. Concludingly, the literature research showed two different approaches to automate architectural security flaw detection. On the one hand, there is work on extracting information from source code or the software's deployment to extract architectural views, such as presented by Abi-Antoun et al., Antonino et al., and Alshammari et al.. On the other hand, some approaches try to automate and improve the detection process of security flaws in architectural views. All mentioned works of the extraction group belong to the detection group, but there is also work solely on the second topic by Sion et al. and Tuma et al.. Their work does not employ static analysis for extracting views, and, consequently, they rely on manually crafted architectural views. As this literature search shows, there is little work on the topic of automatically extracting architectural security views and automatically detecting security flaws. Different works presented different ideas similar to the ones that are

realised in ArchSec. Nevertheless, none of the existing approaches deal with all ideas of ArchSec.

Chapter 14

Conclusions

This chapter recaps the thesis contents and critically discusses the found results. First, Section 14.1 examines the validity of the research hypotheses from Chapter 9 based on the evaluations shown in Chapters 10, 11, and 12. Then, with these results at hand, Section 14.2 answers the proposed research question from Chapter 5 in the scope this thesis provides. Afterwards, Section 14.3 presents the overall contributions of this thesis to the research fields of static analysis, software architecture, and software security. Since the thesis did not enlighten all aspects of the automatic extraction and detection of architectural security flaws, Section 14.4 collects open research questions and shows possibilities for further research topics. Finally, Section 14.5 concludes this chapter and, therefore, the thesis.

14.1. Discussion of Research Hypotheses

As the ArchSec approach consists of an extraction phase and a detection phase, Chapter 9 introduced two primary hypotheses: *"H1—The presented static analysis techniques can correctly extract extended dataflow diagrams for component-based software systems automatically"* and *"H2—The proposed approach can automatically detect present security flaws in extended dataflow diagrams"*. The following subsections discuss the validity of both in greater detail.

14.1.1. The Extraction Hypothesis *H1*

For the first secondary hypothesis, *"H1.1—The presented static analysis techniques can correctly extract architectural views for JavaEE systems automatically"* five different software systems have been analysed, and an extraction has successfully been attempted. These systems range from a small tutorial application to larger commercial applications with over 600k source lines of code. From a technical point of view, all five extraction processes were successful and

were performed in a reasonable amount of time, e.g. less than five minutes for the most extensive application. Considering the accuracy of the extraction result, a manual comparison with application experts in three cases yielded no major complaints. For the other two, no software experts were available due to the open-source nature of the applications. Nevertheless, as the security flaw detection process was successful in all cases—the identified security flaws were present in the actual program—can be taken as an indicator for a successfully extended dataflow diagram extraction.

For the second secondary hypothesis, *"H1.2—The presented static analysis techniques can correctly extract architectural views for Android systems automatically"* ArchSec extracted an extended dataflow diagram for more than 900 Android apps in three different case studies. As there were only binaries available, there is no way to measure the actual size of the apps in source lines of code. However, the size of the analysed bytecode files ranged from several kilobytes to megabytes, which is comparable to the size of the analysed JavaEE programs. Again, the extraction process terminated in a reasonable amount of time for all of them. In the first two case studies, consisting of three apps, an application expert manually checked the extracted diagrams. For the last case study, a sample of 10 randomly chosen apps has been drawn from the entirety of 938 apps. A manual approvement process yielded no relevant differences to manual created extended dataflow diagrams that were created based on the app's decompiled application code.

Overall, one can conclude that the analysis of nearly 1000 component-based software systems yielded an extraction result for each of them. A small subset of these applications was manually checked either by system experts or by the author of this thesis. These checks showed that the extracted extended dataflow diagrams represent the applications but did not precisely match their ideas in all cases. There are two main reasons for the differences. First, a program's structure does not fit the mental model of the security experts. Second, the security experts have trust areas in mind that cannot be found in the code. Of course, this is no proof of the extraction hypothesis being true but an indicator for its validity. The successful detection of existing architectural security flaws is a second indicator for this hypothesis.

14.1.2. The Detection Hypothesis *H2*

Considering the third secondary hypothesis *"H2.1—The proposed approach can automatically detect present authentication-related security flaws in extended dataflow diagrams"*, two extended data flow diagrams, the detection process was performed on, were explicit subject to a search for authentication anti-

patterns. Both case studies employ authentication to protect relevant functionality. ArchSec successfully identified external accesses of these functionalities. Additionally, it detected their protection through correctly implemented authentication patterns. However, one application failed to implement the mitigation for all external interfaces. The detection process identified this missing implementation and thus yielded a security flaw. Applying an authentication-bypass rule to a curated set of 26 manually created dataflow diagrams by Tuma et al. [160] showed a precision of 0.39 and a recall of 0.43.

Following up on the fourth secondary hypothesis, *"H2.2—The proposed approach can automatically detect present authorisation-related security flaws in extended dataflow diagrams"*, both JavaEE applications and Android applications contained authorisation patterns. This time, ArchSec detected a diverging authorisation policy implementation for two case studies. In the two other related case studies, ArchSec found issues with authorisation of a different nature. One, the authorisation implementation took place on the client-side, hence, failing to enforce the authorisation policy if an attacker directly interacted with the server. Second, ArchSec correctly identified an instance of the confused deputy problem. For this pattern, all security flaws found by ArchSec were true positives. Precision and recall were not determined for this research hypothesis.

The fifth secondary hypothesis *"H2.3—The proposed approach can automatically detect present cryptography-related security flaws in extended dataflow diagrams"* bases her assessment on roughly 850 analysed systems. For the three JavaEE systems, the detection identified the correct use of cryptographic measures for all but one instance. Thus, ArchSec successfully identified this one case. Additionally, the detection process observed 161 apps where the designers used HTTP instead of HTTPS. All of these were true positives, making these apps vulnerable. In addition, the analysis of the 26 manually created dataflow diagrams resulted in several security flaws related to this hypothesis. The detection's preciseness was 0.64, and the recall was 0.87.

A proof-of-concept for the sixth secondary hypothesis *"H2.4—The proposed approach can automatically detect present security flaws in extended dataflow diagrams that do not fit into the other categories"* was gained by the positive results of an analysis of four different JavaEE systems and over 30 Android apps. ArchSec automatically identified inter-session dataflows for three JavaEE systems, implicating a breach of confidentiality. A subset of these problems was dynamically validated. ArchSec found evidence for an SQL injection for the fourth JavaEE system, which security experts manually validated. Due to the configuration of the employed database, the system was open for all different kinds of attacks, especially command injections. Additionally, ArchSec de-

tected an activated JavaBridge in 35 of 851 Android apps automatically, which an attacker can use to execute arbitrary OS functionalities. All positive results found during the assessment of this hypothesis were true positive, as manual checks proved. Additionally, the detection of the insufficient auditing rule on the 26 manually created dataflow diagrams yielded a precision of 0.39 and a recall of 0.87.

To sum up, ArchSec correctly identified more than 200 different architectural security flaws in roughly 1000 existing software systems. Furthermore, the detection mechanism has a similar detection rate to a comparable approach while being more flexible in supporting new security features. The precision and recall measurements showed that the results depend on the quality of the rule descriptions, the rule implementation, and the way the dataflow diagrams are created.

14.1.3. Summary

The conducted case studies gave evidence for both primary research hypotheses and showed the technical validity of the ArchSec approach. The manual review of the automatically extracted extended dataflow diagrams only revealed minor discrepancies. The validation of the identified security flaws was successful for the majority of the findings. The reasons for the false positives are mitigations that were outside the analysis scope of ArchSec.

A quantification of false negatives would be necessary to achieve a valid acceptance or rejection of the research hypotheses. There are two main reasons why this was impossible to achieve in the presented evaluation. First, to identify false negatives, a gold standard regarding architectural security flaws must be present, which is unlikely to exist for real-world systems. A manual review might come close but is always reviewer subjective as the ground truth in the comparison study showed. Second, although there are several thorough lists of security flaws, such as CWE and CAPEC, it is unknown and rather unlikely that these are complete. Therefore, false negatives can only ever be determined in relation to a predefined set of classes of security flaws. Nevertheless, although no explicit determination of false negatives has been conducted, the manual reviews with application experts did not reveal false negatives rooted in the ArchSec approach in the conducted case studies.

Another critical factor in this evaluation is the number of analysed systems. There is an obvious imbalance between the number of analysed JavaEE systems and Android apps, which is due to the fact that JavaEE systems are proprietary most of the time and, therefore, usually not accessible for analysis purposes. Furthermore, security is a sensitive aspect for companies. Therefore, they tend

to be more secretive about their systems and possible vulnerabilities stemming from them. The situation is different for Android since most Android apps are aimed at end-users and therefore freely available in their respective app stores.

In general, a proof of a positive statement is empirically impossible. Nevertheless, case studies can give relevant hints on the applicability of an approach. Therefore, despite the evident threats to the validity of this evaluation, the shown results support the presented research hypotheses even if they cannot prove them.

14.2. Discussion of Research Questions

This thesis aims to tackle several research questions from three different areas—ranging from a more formal expressiveness of dataflow diagrams to the development of automatic extraction and detection processes. The last chapters answered these research questions partly through empirical evaluations and partly theoretically through the presentation and discussion of a construction process. This section will discuss the presented research questions from Chapter 5 in greater detail.

14.2.1. More Formal Expressive Dataflow Diagrams

The first research question *"RQ1—How can dataflow diagrams be enhanced to support a more formal expressiveness?"* stemmed from the necessity of a higher degree of formality in dataflow diagrams as stated by Dhillon [63]. Section 7.1 took it on itself to tackle this question and derive a method that allows keeping the simplicity and easy-to-follow structure of dataflow diagrams while adding a degree of formal expressiveness. To that end, the extended dataflow diagrams used in ArchSec are enhanced with a schema that allows adaptability to different use cases. The introduction of a defined schema and the possibility of adding semantical information by supplementing the diagram with attributes lays the base for developing an automatic security flaw detection. Furthermore, Section 7.3 uses attributed typed graphs to precisely define formalised EDFDs. They are similar in their expressiveness to the lowered graphs created from EDFDs during the detection process. As the evaluation showed this process to work smoothly, the approach modelled in ArchSec is one possible answer to *RQ1*.

Answering the second research question *"RQ2—What kind of architectural information and security information is of interest during threat modelling?"* required a deep understanding of software architecture and relevant security goals. During the development of an extraction procedure, it became clear which

architectural information was relevant to a later detection of security flaws, namely components, entry points, and dataflows. Additionally, Section 6.1, together with Chapter 2, led to the conclusion that the authentication pattern, authorisation pattern, and confidentiality through cryptographic means were of interest to gain a helpful threat model.

Both research questions from this topic found their answer through the analysis of requirements. Furthermore, the found solutions were implemented and were shown to be technically feasible during the evaluation of the following two research questions.

14.2.2. Automatic Extraction of Architectural Security Views

Aiming at the third research *"RQ3—What kind of architectural information can be automatically extracted for existing software systems?"*, the goal was to be able to extract all architecture-related information that came up as the answer to *RQ2*.

The extraction process presented in Chapter 6 serves as a constructive approach to answering *RQ3* precisely with the relevant information from *RQ2*. However, a mere statement that this information could be automatically extracted would not be sufficient. Therefore, Chapter 9 developed *H1*, stating that the necessary information could actually be extracted by employing ArchSec's extraction process. The evaluation in Chapters 10 and 11 supported this hypothesis, answering the research question as follows: Components, entry points, dataflows, data exits, and trust areas can automatically be extracted for component-based software systems.

The fourth research question *"RQ4—Which architectural security information can be automatically extracted using static software analysis?"* had to be handled comparingly. The answer to *RQ2* identified desirable architectural security information for a threat model, therefore stating the need to extract them automatically. To actually show that this information can be extracted automatically, an evaluation was necessary. Hence, *RQ2* was formulated and differentiated according to the different types of architectural security information. The evaluation presented in Chapters 10 and 11 and its summary in Section 14.1 gave strong evidence for the correctness of this hypothesis and found additional architectural security information that could be extracted using the ArchSec approach. This result yields the following answer to *RQ4*: The ArchSec approach can automatically extract the usage of authentication, authorisation, and encryption. Since ArchSec focuses on detecting the usage of specific security pattern implementations provided by the component-based frameworks, its extraction capability is currently restricted to those. However, a thourough

evaluation of soundness and completeness of the extraction was not conducted, although the studies show exemplary successes.

Summing up, the requirements put up through the answer for *RQ2* came to their fulfilment with the evaluation-based answers for *RQ3* and *RQ4*. In conclusion, it can be stated that the brought-up architectural and security information can automatically and correctly be extracted using ArchSec.

14.2.3. Automatic Detection of Security Flaws

For the answer of *"RQ5—How can we describe architectural security flaws for dataflow diagrams?"*, Chapter 8 introduced a knowledge base designed to capture all necessary information on architectural security flaws. This information consists of a risk assessment, a categorisation, a thorough description of the flaw, and possible mitigation. Additionally, to enable the automatic detection of these flaws, using a graph query language proves to be advantageous. For a given architectural security flaw, the graph query language enables a formal description of the security anti-pattern and the mitigation pattern, if present. The choice of graph query language, as well as the amount of information stored for one security flaw, is a possible answer to *RQ5*.

Turning back to the question of applicability, *"RQ6—How can instances of the architectural security flaw descriptions be automatically identified?"* aims at the incorporation of flaw detection into an automated process. Sections 8.3 and 8.4 described the ArchSec approach to an automatic detection process. First, the extracted extended dataflow diagram is lowered to a property graph, enabling the security anti-patterns to be queried. Second, for any matched anti-pattern, the existence of mitigation patterns is determined. Analogously to *RQ3* and *RQ4*, a simple construction of a detection process does not suffice as a confident answer. Therefore, the research hypotheses developed in Chapter 9 received an empirical evaluation. Automatically extracted (c.f. Chapters 10 and 11) as well as manually created (c.f. Chapter 12) EDFDs have been subjected to the security flaw detection process of ArchSec. In most cases, ArchSec managed to identify present security flaws and mitigations correctly. When compared to a different automatic detection approach, ArchSec performed comparably well and slightly better for certain instances. Therefore, *RQ6* can be answered as follows: Building on an EDFD, a graph lowering allows the automatic detection of security flaws that have formally been defined in a graph query language.

All in all, the formal expression of security flaws as subgraphs together with the realisation of EDFDs as property graphs reduces the task of automatically detecting architectural security flaws to the subgraph isomorphism problem.

Several case studies showed the applicability of this approach.

14.2.4. Summary

Together with the empirical evaluation, the theoretical groundwork was able to answer all six research questions posed in Chapter 5. Therefore, ArchSec is a step towards automated threat modelling but undoubtedly not the only one. Especially, *RQ1* and *RQ5* have many different potential solutions, which would, of course, influence the answer for *RQ6* as well. The next section will discuss more details on this.

14.3. Synopsis

Architectural security flaws are more and more in the attention of attackers. Therefore, this thesis proposed an approach dealing with their automatic detection. As presented in the Prelude, architectural security flaws violate information security's protection goals and deal with authorisation, authentication, and cryptography. The proposed ArchSec approach consists of two analysis parts. First, it uses static analysis techniques to extract extended dataflow diagrams automatically. Extended dataflow diagrams are a proposed and more semantic extension of traditional dataflow diagrams. Then an automated security flaw detection works on these diagrams and uses a knowledge base that encodes architectural security (anti-)patterns. A domain-specific language describes the security flaw rules stored in the knowledge base. The concrete patterns are noted using Cypher, a graph query language. The result of this automated process is a threat model that security experts, software architects, developers and quality assurance can use as a reference for securing the software under investigation. The development of ArchSec set out to address several problems that arise with the assessment of a software's security. These problems come from different research topics and have been investigated in three different research areas, each being specified by two research questions.

In the Prelude, it was pointed out that there is a dire need of including security information in architectural views. With the development of extended dataflow diagrams, answering *RQ1* and *RQ2*, it became possible to formally include security information into architectural views, thus building a document that can automatically be extracted and serve as a base for the automatic detection of security flaws. This extension aims at the intersection between security and software architecture, thus tackling research area *T1*.

Although the chosen extended dataflow diagrams offer a good starting point for architecturally modelling security aspects, there are some security aspects

that they cannot express. An essential aspect security experts look at during their assessments are security protocols. Since extended dataflow diagrams have currently no means to express the ordering of information flows, one solution would be to enhance them using the attribute mechanism. However, this approach is not practical since the chosen graph query language cannot explicitly express time constraints. Another possible solution to this problem would have been to select other modelling approaches, such as UML accompanied by OCL constraints. While this is undoubtedly a valid approach, there are reasons for not having chosen this way. First, UML splits structural and behavioural aspects of a system into different views, making it difficult for a human being to analyse them as a whole. Second, during its evolution process, ArchSec used OCL in the first place for expressing security rules. Still, the language turned out to have a limited expressiveness for matching paths of an arbitrary length, which is essential for some types of flaws.

Additionally, the probably existing difference between every planned architecture and implemented architecture leads to the conclusion that it is less helpful to analyse the planned architecture when the implemented one is the one in action. Therefore, *RQ3*'s and *RQ4*'s answers and evaluation provided a crucial step forward, enabling security experts to analyse an architectural view that represents the actual implementation of the software system. Furthermore, as the proposed approach uses static analysis techniques to extract EDFDs for software systems, it belongs to the intersection of *static analysis* and *software architecture*, tackling research area *T2*.

A very fundamental design decision was to focus on component-based software frameworks in the EDFD extraction process. One could argue that choosing a pattern-based architecture recovery approach, as presented in the literature on software architecture recovery, without special knowledge of component-based software systems would have been sufficient. The implemented architectural component detection actually is a pattern-based approach since the detection searches for classes adhering to a specific implementation pattern. The advantage of building on top of the used frameworks is the possibility of identifying component boundaries and component entries and exits, which are relevant for threat modelling. Another decision that influenced ArchSec's design was to focus on the three security patterns authentication, authorisation, and cryptography. This decision influenced the formalisation of extended dataflow diagrams. As already discussed, ArchSec cannot check for protocol-related security flaws at the level of EDFDs. In this context, protocol-related means that a security protocol—the timely ordered exchange of messages, for instance, the TLS handshake for determining the used encryption algorithm—may result in an insecure state. For this purpose, security research employs, for instance,

model checking [25]. This shortcoming finds its reason in focusing on the aspects of information security, where the three patterns mentioned above solve the violation of information security goals.

Finally, some false positives occurred in the evaluation of the security flaw detection process. Looking closer at these false positives revealed that the common cause of the false positives was the missing knowledge of trust boundaries. Since these trust boundaries were not part of the implementation, the extraction process could not capture them. Please see the next section on open research topics for an idea of how to deal with these.

Both phases of ArchSec face their own challenges regarding the validity of their results. The extraction phase is a static analysis and therefore the main objective is completeness. However, there are different aspects within software systems that static analysis cannot rule out statically in all cases, e.g., it is impossible to determine all the opened files or accessed URIs since some of the accessed files or URIs depend on dynamic information. Therefore, it is necessary to show that the file or URI is constructed at runtime. In such a case, an unknown file or URI is generated in the extended dataflow diagram if the extraction cannot determine the exact file to model the possible threat of an attacker opening arbitrary files. As already mentioned, a proper determination of precision and recall for the extraction process was not conducted. In fact, the missing precision of the extraction influences the precision of the following detection process, in addition to the possible false positives of the detection process itself. In the above example, a manual security review is still necessary to identify possibly existing mitigations to this potential threat. Besides the discussed problem arising from the extraction phase, security detection has to deal with additional security-related challenges. First, it is important to state that a perfectly secure system does not exist, and literature states that one has to deal with secure enough systems [85, 138, 156]. Since it is never certain that all security flaws are known, an objectively perfect recall cannot be achieved. Standard evaluations compare to manual security assessments. Humans, however, are not perfect. Security assessments are usually conducted in a more or less ad-hoc fashion and are thus error-prone. Additionally, different experts may interpret security requirements differently. Thus, the primary goal of ArchSec is to automate and objectify the process. Concludingly, it cannot prove the absence of security flaws but help with the identification of such.

Additionally, a point of criticism can be that ArchSec focuses on the usage of security libraries and the framework's security means. It does not support application-specific security mechanisms. It is possible to develop static analyses that support application-specific security means adding additional information to the extracted extended dataflow diagram. But, a very fundamental principle

of software security is to use security standards [52] since many security experts reviewed them. Manual security mechanisms require an intensive review by security experts to ensure their correct functionality. Therefore, ArchSec would report missing security measures for manually crafted security mechanisms, resulting in an inspection of the application-specific security.

Reminding oneself that the initial goal was to make software systems more secure, one key aspect may not be forgotten: The assessment of an architectural view—how elaborate it may be—is a very subjective process, thus highly dependent on the expert's knowledge and experience as well as her availability. Unfortunately, not every software system is maintained by a company having the resources to employ a security expert nor a static analysis expert. ArchSec solves this problem by automating the architectural view creation using architecture recovery incorporating knowledge of static analysis experts. Furthermore, ArchSec's extensible knowledge base—storing known security flaws with a detailed description and risk assessment, as well as potential mitigations—allows security, domain, and technical experts to provide their knowledge without being familiar with every system's quirks and specific attack scenarios. Therefore, ArchSec does both: It maintains a knowledge base to profit from the experience of more than one security expert, and it formalises the description stored in the database such that automatic detection of security flaws becomes possible. Both research questions in this regard could be answered, and—where necessary—evaluated. Therefore, the automatic detection of ArchSec successfully poses a significant extension to the field of research between security and static analysis, that is, research area *T3*.

The introduction mentioned that a disadvantage of the manual approach is its subjectivity. One could also raise this argument for the ArchSec approach. While this is true in the current situation as there are only rules created by the thesis' author, the introduced knowledge base allows the addition of new or the adaption of old security flaw rules by other researchers and security experts, thus levelling out the subjective point-of-view the thesis's author might have. Since the current knowledge base was extended step-by-step, as more and more case studies were analysed, it is a valid critique of the knowledge base's transferability and the generality of the evaluation that the test set equals the training set. A thorough evaluation of the ArchSec approach using the current knowledge base is therefore necessary with a new set of software systems.

A problem of the presented approach is that the security means are detected separately. In a very rare situation, two mitigated flaws can be classified as such, although their combination remains insecure and undetected. This can happen, for instance, with the combination of message encryption and channel encryption. Moreover, the combination of multiple encryption schemes may

allow an attacker to derive the content more easily than to encrypt it once. Nevertheless, it is possible to write additional rules to identify such security anti-patterns and mark occurrences in extended dataflow diagrams.

14.4. Open Research Topics

Since no research can shed light on all aspects of a problem and answer all possible research questions in a finite amount of time, the remainder of this section lists several open research topics, which might interest other PhD students and researchers. The order of the given research topic does neither correspond to their importance nor to any other metric.

The extended dataflow diagram extraction generates an artificial main method for statically analysing a software system. In total, this step consists of two phases. First, static analysis techniques are used to identify existing components in an applications source code. Second, the gained information is used to generate the artificial main method. The downside of this approach is the engineering effort necessary to support new frameworks. An improvement of this idea would be not to generate a main method but to generate intermediate information on instantiated objects, possible references between these objects by storing points-to sets, and object process graphs for recording object lifecycles. Finally, existing callgraph construction algorithms and pointer analyses would be needed to be adapted to take this information into account. This approach would reduce the amount of work necessary to support new frameworks and allow adding support for frameworks such as CDI. Thus, many analyses would profit from such a framework.

The currently implemented extraction-related analyses focus on the structure of the analysed software systems and used security features. Nevertheless, there is a prototype for adding actual dataflow information to an extended dataflow diagram. The prototype takes a user-given configuration, performs a rudimentary inter-procedural dataflow analysis, and adds this information to an EDFD. Sadly, this information is not integrated with the information resulting from the static analyses dealing with software security means. Such a combination would result in new categories of security flaws to detect, such as the policy-wise protection of information flows, instead of pure access to sensitive resources.

Furthermore, a more thourough evaluation of all phases of ArchSec would be beneficial. At the extraction level it is necessary to determine precision and recall for different static analysis components, e.g., the architecture recovery and the various security fact extractions. For this purpose, it would be neces-

sary to manually create a ground truth for each evaluation aspect. Another possibility is the usage of dynamic analyses to validate the static results, for instance, by monitoring the opened files of an application and comparing those to the statically identified set. At the detection level, an evaluation of higher sample size could underline or refute the potential of ArchSec's security flaw detection.

Currently, the created extended dataflow diagrams focus on the analysed software system and not its environment. However, using network scanners, it would be possible to automatically detect security mechanisms outside the application, such as firewalls or HTTP proxies. Firewalls can protect services from being accessible from the world wide web, thus possible protecting parts of the application. HTTP proxies are frequently used to encrypt outgoing and incoming traffic. Currently, the ArchSec analysis would detect that the application is sending sensitive information without any encryption, but the HTTP proxy uses TLS as transport encryption. A comparable approach has been proposed by Välja et al. [166]. An additional analysis of the deployment environment would allow removing such, from a user's point of view, false alarms.

14.5. Chapter Summary

This chapter brought together all contributions presented in this thesis. To that final end, it restated the problem discussed in the introduction, the research areas this thesis contributed to, the research questions answered along this way, and the research hypotheses that were supported through the evaluations. It identified possible criticisms and either invalidated them, argued them to be design decisions, or marked them for further research. Additionally, it gave several topics for aspiring PhD students to delve into for their own research.

Part V

Appendix

Appendix A

ArchSec's Core Schema

This chapters shows ArchSec's core built-in schema. It is not necessary to understand the thesis' content is just printed for informative purposes.

```
schema {
  types {
    ChannelType "Communication" {
      children {
        ChannelType "Interprocess" {
          children {
            ChannelType "Email";
            ChannelType "Http" {
              children {
                ChannelType "Https" {
                  implies {"Encryption.Transport.TLS" : true}
                };
              }
            };

            ChannelType "RPC";
          }
        };

        ChannelType "IntraProcess";
      }
    };

    ChannelType "Manual Input";

    ElementType "Data Store" {
      children {
```

```
    ElementType "Directory";
    ElementType "File";
  }
};

ElementType "Device" {
  children {
    ElementType "Server";
    ElementType "Desktop";
  }
};

ElementType "External Entity" {
  children {
    ElementType "Subject";
  }

  implies { "external" : true }
};

ElementType "Process";

ElementType "Software" {
  children {
    ElementType "Library";
    ElementType "Component";
  }
};

DataType "System Data";

DataType "User Data" {
  children {
    DataType "Personal Information" {
      children {
        DataType "Credentials";
      }
      implies {"Security Objective.Confidentiality" : true}
    };
```

```
    }
  };

  TrustAreaType "Device";
  TrustAreaType "Network";
}

attributes {
  ChannelAttribute "Encryption" of type Boolean {
    children {
      ChannelAttribute "Message" of type Boolean;
      ChannelAttribute "Transport" of type Boolean {
        children {
          ChannelAttribute "TLS" of type Boolean;
        }
      };
    }
  };

  ChannelAttribute "UID" of type String;
  ElementAttribute "UID" of type String;
  DataAttribute "UID" of type String;
  TrustAreaAttribute "UID" of type String;

  DataAttribute "Security Objective.Confidentiality" of type
  ↪  Boolean;
  DataAttribute "Security Objective.Integrity" of type Boolean;
  DataAttribute "Security Objective.NonRedpudiation" of type
  ↪  Boolean;
  ElementAttribute "Security Objective.Availability" of type
  ↪  Boolean;
  ElementAttribute "Security Objective.NonRedpudiation" of type
  ↪  Boolean;
  ElementAttribute "Security Objective.Authenticity" of type
  ↪  Boolean;
  ChannelAttribute "Security Objective.Authenticity" of type
  ↪  Boolean;

  ElementAttribute "external" of type Boolean;
```

```
  }
}
```

Appendix B

List of Supported CWE Entries and CAPEC Entries

ArchSec supports different CWE entries and CAPEC entries. All entries together with their short description are listed in the following:

CWE-22: Improper Limitation of a Pathname to a Restricted Directory

CWE-78: Improper Neutralisation of Special Elements used in an OS Command

CWE-79: Improper Neutralisation of Input During Web Page Generation

CWE-89: Improper Neutralisation of Special Elements used in an SQL Command

CWE-120: Buffer Copy without Checking Size of Input

CWE-134: Uncontrolled Format String

CWE-190: Integer Overflow or Wraparound

CWE-288: Authentication Bypass Using an Alternate Path or Channel

CWE-306: Missing Authentication for Critical Function

CWE-311: Missing Encryption of Sensitive Data

CWE-319: Cleartext Transmission of Sensitive Information

CWE-327: Use of a Broken or Risky Cryptographic Algorithm

CWE-352: Cross-Site Request Forgery (CSRF)

CWE-602: Client-Side Enforcement of Server-Side Security

CWE-759: Use of a One-Way Hash without a Salt

CAPEC-16: Dictionary-based Password Attack

CAPEC-22: Exploiting Trust in Client (aka Make the Client Invisible)

CAPEC-66: SQL Injection

CAPEC-94: Man in the Middle Attack

CAPEC-108: Command Line Execution through SQL Injection

Appendix C

Excerpt from *A Catalog of Design Flaws*

This chapter lists the design flaw descriptions given by Tuma et al. [160] used in the comparison study to create ArchSec's detection rules.

C.1. Flaw 2 – Authentication Bypass Using an Alternate Path

Description

This refers to the case where although there is an authentication mechanism in place, it does not cover all possible entry points to the system. This can be due to the fact that there is a remote access point to the system aiming towards support or maintenance, a possible backdoor. Additionally, a system may make a call to invoke functionality of an external application. In such case, the external application can have access to resources and data of the system if not contained properly.

Detection

- Determine the entry points to the system.
- for each entry point examine:
  - Does it go through an authentication point?
  - What kind of assets are accessible through this path? Are their security objectives still achieved?
  - Is the system protected against MITM and session hijacking attacks? Are the communication channels used encrypted?

  - Does the system invoke functionality from third party applications? Are these applications subject to proper access control (regarding the resources and data they have access to)?

C.2. Flaw 6 – Insufficient Cryptographic Keys Management

Description

This refers to the generation, distribution and storage of the cryptographic keys in the system. A compromise of a key means that every piece of information encrypted with this key is compromised. If there is no mechanism to replace a key or any auditing to aid in recovering, this can lead to serious compromise of the system.

Detection

- Determine where cryptographic keys are created, stored, used and how they are distributed.
- Identify the mechanisms in place to replace a key.
- Track where the keys are used, for which purpose and who has access to them.
- For each key examine:
  - Is it generated within a cryptographic module isolated from the rest of the system?
  - Does the Random Number Generator (RNG) comply with latest standards?
  - Is the time the key is in plaintext format minimized?
  - Is the access to it during that time restricted only to authorized parties?
  - Are the keys distributed through secure channels?
  - Is the key stored securely?
  - If the key is stored locally in devices/clients, is it encrypted with Key Encryption Key (KEKs)?
  - Is the key integrity ensured?

  - Is the key used only by processes within the cryptographic module?
  - Is there a secure backup of the datastore that stores the keys?
  - Is all access to the key in plaintext format logged?
  - Is a key only used for a single purpose? (For example only for encrypting data but not other keys)
  - Is the key destroyed after it is no longer needed?
  - Is there a mechanism in place to renew/replace the keys in case they are compromised?

C.3. Flaw 13 – Insecure Data Storage

Description

Data stored on system are not sufficiently protected. This involves data stored in a resource, which could be a database, memory, temporary files, cookies etc., that are stored in cleartext format or can be accessed by users or processes that should not have permissions to do so. Additionally, sensitive data may be stored along with the rest of the data in one place.

Detection

- Determine the sensitive data stored in the system.
- Locate where these data are stored.
- Determine the actors and processes that can access them. • For each case examine:
  - Are these data deleted when they are no longer needed?
  - Are these data encrypted while stored?
  - Is there an access control mechanism in place to prevent unauthorised access?
  - Is any data that come from computations or processing of sensitive data protected as well?
  - Is access to data logged? (actor, timestamp etc)
  - Is there a backup for sensitive data?
  - Are the backups stored in different physical devices?
  - Is data separated according to their sensitivity level?

C.4. Flaw 15 – Insecure Data Exposure

Description

Data is not transferred in a secure way. For example web application uses HTTP instead of HTTPS. This leaves the channel vulnerable to eavesdropping, Man In The Middle (MITM) attacks etc.

Detection

- Locate the valuable information in the model.
- Track them through the architecture to determine where and how they are transferred.
- At each step examine the following:
  - Is the reuse of packets prevented (Replay attacks)?
  - Is there any form of timestamping, message sequencing or checksum in the exchanged packages?
  - Is the traffic over an encrypted channel (SSL/TLS)?

C.5. Flaw 18 – Insufficient Auditing

Description

Access to critical resources or operations is not logged. This can lead to repudiation issues and possibly makes it more difficult to recover after an attack.

Detection

- Determine for which events or operations it is important to log information in the system
- For each case examine:
  - Is the access to sensitive data and operations logged?
  - Are changes in privileges logged?
  - Are failed attempts to authenticate or access a resource logged?
  - Do the logs keep information about time, IP, ID when an actor performs an operation?

- Is the information logged sufficient to trace where/when an issue occurred and what was altered/corrupted?
- Are these logs stored securely?
- Is there any backup of the logs in case they are erased by the attacker?

Appendix D

Detailed Comparison Results

This chapter presents the detailed results of the comparison presented in Section 12.3.

Table D.1. Detailed results for *Security Flaw 2* for both approaches.

		BeSocial	*DriveSafe*	*PhotoFriends*	*SmartTex*	*Total*
H2.1						
Flaw 2						
ArchSec						
	TP	4	2	6	11	*23*
	FP	11	8	9	8	*36*
	FN	6	7	7	10	*30*
	precision	*0.27*	*0.20*	*0.40*	*0.58*	*0.39*
	recall	*0.40*	*0.22*	*0.46*	*0.52*	*0.43*
	F1-measure	*0.32*	*0.21*	*0.43*	*0.55*	*0.41*
Tuma						
	TP	4	2	9	13	*28*
	FP	15	14	10	14	*53*
	FN	6	7	5	10	*28*
	precision	*0.21*	*0.13*	*0.47*	*0.48*	*0.35*
	recall	*0.40*	*0.22*	*0.64*	*0.57*	*0.50*
	F1-measure	*0.28*	*0.16*	*0.57*	*0.52*	*0.41*

Table D.2. Detailed results for *Security Flaw 6* for both approaches.

		BeSocial	*DriveSafe*	*PhotoFriends*	*SmartTex*	*Total*
H2.3						
Flaw 6						
ArchSec						
	TP	14	8	20	15	*57*
	FP	12	7	18	10	*47*
	FN	0	0	2	2	*4*
	precision	*0.54*	*0.53*	*0.68*	*0.60*	*0.55*
	recall	*1.00*	*1.00*	*0.91*	*0.88*	*0.93*
	F1-measure	*0.84*	*0.63*	*0.78*	*0.71*	*0.69*
Tuma						
	TP	16	5	21	14	*56*
	FP	6	6	10	13	*35*
	FN	0	0	2	2	*4*
	precision	*0.73*	*0.45*	*0.68*	*0.52*	*0.62*
	recall	*1.00*	*1.00*	*0.91*	*0.88*	*0.93*
	F1-measure	*0.84*	*0.63*	*0.78*	*0.65*	*0.74*

Table D.3. Detailed results for *Security Flaw 13* for both approaches.

		BeSocial	*DriveSafe*	*PhotoFriends*	*SmartTex*	*Total*
H2.3						
Flaw 13						
ArchSec						
	TP	56	26	29	48	*159*
	FP	16	28	24	2	*60*
	FN	4	6	12	5	*27*
	precision	*0.78*	*0.59*	*0.55*	*0.96*	*0.73*
	recall	*0.93*	*0.81*	*0.71*	*0.91*	*0.85*
	F1-measure	*0.85*	*0.68*	*0.62*	*0.93*	*0.79*
Tuma						
	TP	26	13	14	23	*76*
	FP	4	4	4	1	*13*
	FN	4	6	12	5	*27*
	precision	*0.87*	*0.76*	*0.78*	*0.96*	*0.85*
	recall	*0.87*	*0.68*	*0.54*	*0.82*	*0.74*
	F1-measure	*0.87*	*0.72*	*0.64*	*0.88*	*0.79*

Table D.4. Detailed results for *Security Flaw 15* for both approaches.

		BeSocial	*DriveSafe*	*PhotoFriends*	*SmartTex*	*Total*
H2.3						
Flaw 15						
ArchSec						
	TP	40	47	52	26	*165*
	FP	31	19	23	36	*109*
	FN	2	3	12	7	*24*
	precision	*0.56*	*0.71*	*0.69*	*0.42*	*0.60*
	recall	*0.95*	*0.94*	*0.81*	*0.79*	*0.87*
	F1-measure	*0.71*	*0.81*	*0.75*	*0.55*	*0.71*
Tuma						
	TP	40	47	51	25	*163*
	FP	40	27	30	50	*147*
	FN	2	2	12	7	*23*
	precision	*0.50*	*0.64*	*0.63*	*0.33*	*0.53*
	recall	*0.95*	*0.96*	*0.81*	*0.78*	*0.88*
	F1-measure	*0.66*	*0.76*	*0.71*	*0.47*	*0.66*

Table D.5. Detailed results for *Security Flaw 18* for both approaches.

		BeSocial	*DriveSafe*	*PhotoFriends*	*SmartTex*	*Total*
H2.4						
Flaw 18						
ArchSec						
	TP	7	10	11	11	*39*
	FP	24	12	14	11	*61*
	FN	3	2	0	1	*6*
	precision	*0.22*	*0.45*	*0.44*	*0.50*	*0.39*
	recall	*0.70*	*0.83*	*1.00*	*0.92*	*0.87*
	F1-measure	*0.34*	*0.59*	*0.61*	*0.65*	*0.54*
Tuma						
	TP	2	0	4	2	*8*
	FP	6	8	3	5	*22*
	FN	6	6	1	4	*17*
	precision	*0.25*	*0.00*	*0.57*	*0.29*	*0.27*
	recall	*0.25*	*0.00*	*0.80*	*0.33*	*0.32*
	F1-measure	*0.25*	*0.00*	*0.67*	*0.31*	*0.29*

Bibliography

[1] Regulation (eu) 2016/679 of the european parliament and of the council of 27 april 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing directive 95/46/ec (general data protection regulation).

[2] MIL-STD-882C – System Safety Program Requirements. Technical report, Department of Defense, January 1993.

[3] ISO 9000:2015(en) Quality management systems — Fundamentals and vocabulary. Technical report, International Organization for Standardization, 2015.

[4] Common Attack Pattern Enumeration and Classification. online, MITRE Corporation, May 2021. https://capec.mitre.org.

[5] Common Weakness Enumeration. online, MITRE Corporation, May 2021. https://cwe.mitre.org.

[6] ENNode. online, Environmental Council of the States, May 2021. https://archive.codeplex.com/?p=ennode.

[7] Gss. online, Electronic Business Systems, Ltd, May 2021. https://code.google.com/archive/p/gss/.

[8] Marwan Abi-Antoun and Jeffrey M. Barnes. Analyzing security architectures. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, page 3–12, New York, NY, USA, 2010. Association for Computing Machinery.

[9] Marwan Abi-Antoun, Daniel Wang, and Peter Torr. Checking threat modeling data flow diagrams for implementation conformance and security. In *Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, page 393–396, New York, NY, USA, 2007. Association for Computing Machinery.

[10] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA, 2006.

[11] A. Alkussayer and W. H. Allen. Security risk analysis of software architecture based on ahp. In *7th International Conference on Networked Computing*, pages 60–67, 2011.

[12] Hussain Almohri, Long Cheng, Danfeng Yao, and Homa Alemzadeh. On threat modeling and mitigation of medical cyber-physical systems. In *2017 IEEE/ACM International Conference on Connected Health: Applications, Systems and Engineering Technologies (CHASE)*, pages 114–119, 2017.

[13] M. Almorsy, J. Grundy, and A. S. Ibrahim. Supporting automated vulnerability analysis using formalized vulnerability signatures. In *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 100–109, 2012.

[14] Mohamed Almorsy, John Grundy, and Amani S. Ibrahim. Automated software architecture security risk analysis using formalized signatures. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, page 662–671. IEEE Press, 2013.

[15] B. Alshammari, C. Fidge, and D. Corney. A hierarchical security assessment model for object-oriented programs. In *2011 11th International Conference on Quality Software*, pages 218–227, July 2011.

[16] B. Alshammari, C. Fidge, and D. Corney. Security assessment of code refactoring rules. In *WIAR 2012; National Workshop on Information Assurance Research*, pages 1–10, April 2012.

[17] American National Standards Institute. Role based access control. Technical report, 2004. ANSI/INCITS 359-2004.

[18] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan Reutter, and Domagoj Vrgoč. Foundations of modern query languages for graph databases. *ACM Comput. Surv.*, 50(5), September 2017.

[19] Pablo Antonino, Slawomir Duszynski, Christian Jung, and Manuel Rudolph. Indicator-based architecture-level security evaluation in a service-oriented environment. In *Proceedings of the Fourth European Conference on Software Architecture: Companion Volume*, ECSA '10, page 221–228, New York, NY, USA, 2010. Association for Computing Machinery.

[20] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *SIGPLAN Not.*, 49(6):259–269, June 2014.

[21] Dejan Baca and Bengt Carlsson. Agile development with security engineering activities. In *Proceedings of the 2011 International Conference on Software and Systems Process*, ICSSP '11, page 149–158, New York, NY, USA, 2011. Association for Computing Machinery.

[22] David F. Bacon and Peter F. Sweeney. Fast static analysis of c++ virtual function calls. *SIGPLAN Notices*, 31(10):324–341, October 1996.

[23] Steffen Bartsch, Bernhard J. Berger, Eric Bodden, Achim D. Brucker, Jens Heider, Mehmet Kus, Sönke Maseberg, Karsten Sohr, and Melanie Volkamer. Zertifizierte datensicherheit für android-anwendungen auf basis statischer programmanalysen. In Stefan Katzenbeisser, Volkmar Lotz, and Edgar R. Weippl, editors, *Sicherheit 2014: Sicherheit, Schutz und Zuverlässigkeit, Beiträge der 7. Jahrestagung des Fachbereichs Sicherheit der Gesellschaft für Informatik e.V. (GI), 19.-21. März 2014, Wien, Österreich*, volume P-228 of *LNI*, pages 283–291. GI, 2014.

[24] Steffen Bartsch, Bernhard J. Berger, Michaela Bunke, and Karsten Sohr. The transitivity-of-trust problem in android application interaction. In *2013 International Conference on Availability, Reliability and Security, ARES 2013, Regensburg, Germany, September 2-6, 2013*, pages 291–296. IEEE Computer Society, 2013.

[25] David Basin, Cas Cremers, and Catherine Meadows. *Model Checking Security Protocols*, pages 727–762. Springer International Publishing, Cham, 2018.

[26] David A. Basin, Manuel Clavel, Jürgen Doser, and Marina Egea. Automated analysis of security-design models. *Inf. Softw. Technol.*, 51(5):815–831, 2009.

[27] David A. Basin, Jürgen Doser, and Torsten Lodderstedt. Model driven security: From UML models to access control infrastructures. *ACM Trans. Softw. Eng. Methodol.*, 15(1):39–91, 2006.

[28] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley Professional, 3rd edition, 2012.

[29] Len Bass, Ralph Holz, Paul Rimba, An Binh Tran, and Liming Zhu. Securing a deployment pipeline. In *Proceedings of the Third International Workshop on Release Engineering*, RELENG '15, page 4–7. IEEE Press, 2015.

[30] Anika Behrens. What are security patterns? a formal model for security and design of software. In *Proceedings of the 13th International Conference on Availability, Reliability and Security*, ARES 2018, New York, NY, USA, 2018. Association for Computing Machinery.

[31] Bernhard J. Berger, Michaela Bunke, and Karsten Sohr. An android security case study with bauhaus. In Martin Pinzger, Denys Poshyvanyk, and Jim Buckley, editors, *18th Working Conference on Reverse Engineering, WCRE 2011, Limerick, Ireland, October 17-20, 2011*, pages 179–183. IEEE Computer Society, 2011.

[32] Bernhard J. Berger, Christian Maeder, Rodrigue Wete Nguempnang, Karsten Sohr, and Carlos E. Rubio-Medrano. Towards effective verification of multi-model access control properties. In Florian Kerschbaum, Atefeh Mashatan, Jianwei Niu, and Adam J. Lee, editors, *Proceedings of the 24th ACM Symposium on Access Control Models and Technologies, SACMAT 2019, Toronto, ON, Canada, June 03-06, 2019*, pages 149–160. ACM, 2019.

[33] Bernhard J. Berger, Christian Maeder, and Salva Daneshgadeh Çakmakçı. Threat modeling knowledge for the maritime community. In *Proceedings of the first European Workshop on Maritime Systems Resilience and Security (MARESEC 2021)*, 2021. accepted for publication.

[34] Bernhard J. Berger, Rodrigue Wete Nguempnang, Karsten Sohr, and Rainer Koschke. Static extraction of enforced authorization policies – seeauthz. In *2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 187–197, 2020.

[35] Bernhard J. Berger and Karsten Sohr. An approach to detecting inter-session data flow induced by object pooling. In Dimitris Gritzalis, Steven Furnell, and Marianthi Theoharidou, editors, *Information Security and Privacy Research - 27th IFIP TC 11 Information Security and Privacy Conference, SEC 2012, Heraklion, Crete, Greece, June 4-6, 2012. Proceedings*, volume 376 of *IFIP Advances in Information and Communication Technology*, pages 25–36. Springer, 2012.

[36] Bernhard J. Berger, Karsten Sohr, and Udo H. Kalinna. Architekturelle Sicherheitsanalyse für Android. In Patrick Horster, editor, *D • A • CH Security 2014: Bestandsaufnahme - Konzepte - Anwendungen - Perspektiven*, pages 287–298. Peter Schartner and Peter Lipp, 2014.

[37] Bernhard J. Berger, Karsten Sohr, and Rainer Koschke. Extracting and analyzing the implemented security architecture of business applications. In Anthony Cleve, Filippo Ricca, and Maura Cerioli, editors, *17th European Conference on Software Maintenance and Reengineering, CSMR 2013, Genova, Italy, March 5-8, 2013*, pages 285–294. IEEE Computer Society, 2013.

[38] Bernhard J. Berger, Karsten Sohr, and Rainer Koschke. Automatically extracting threats from extended data flow diagrams. In Juan Caballero, Eric Bodden, and Elias Athanasopoulos, editors, *Engineering Secure Software and Systems - 8th International Symposium, ESSoS 2016, London, UK, April 6-8, 2016. Proceedings*, volume 9639 of *Lecture Notes in Computer Science*, pages 56–71. Springer, 2016.

[39] Bernhard J. Berger, Karsten Sohr, and Rainer Koschke. The architectural security tool suite - ARCHSEC. In *19th International Working Conference on Source Code Analysis and Manipulation, SCAM 2019, Cleveland, OH, USA, September 30 - October 1, 2019*, pages 250–255. IEEE, 2019.

[40] Bastian Best, Jan Jürjens, and Bashar Nuseibeh. Model-based security engineering of distributed information systems using umlsec. In *29th International Conference on Software Engineering (ICSE'07)*, pages 581–590, 2007.

[41] Eric Bodden. Inter-procedural data-flow analysis with ifds/ide and soot. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program Analysis*, SOAP '12, pages 3–8, New York, NY, USA, 2012. Association for Computing Machinery.

[42] B. Boehm and R. Turner. Management challenges to implementing agile processes in traditional development organizations. *IEEE Software*, 22(5):30–39, 2005.

[43] Barry Boehm and Victor R. Basili. Software defect reduction top 10 list. *Computer*, 34(1):135–137, January 2001.

[44] Alexandre M. Braga, Cecilia M. F. Rubira, and Ricardo Dahab. Tropyc: A pattern language for cryptographic software. In *Proceedings of the Pattern Languages of Programs Conference*, 1998.

[45] Michaela Bunke. *Security-Pattern Recognition and Validation*. PhD thesis, University of Bremen, Germany, 2019.

[46] Michaela Bunke and Karsten Sohr. An architecture-centric approach to detecting security patterns in software. In Úlfar Erlingsson, Roel Wieringa, and Nicola Zannone, editors, *Engineering Secure Software and Systems*, pages 156–166, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[47] Michaela Bunke and Karsten Sohr. Towards supporting software assurance assessments by detecting security patterns. *Software Quality Journal*, 28(4):1711–1753, December 2020.

[48] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. Wiley, Chichester, UK, 1996.

[49] Andrew Butterfield, Gerard Ekembe Ngondi, and Anne Kerr, editors. *A Dictionary of Computer Science*. Oxford University Press, 7th edition, 2016.

[50] Shing Wai Chan and Ed Burns. Java™ servlet specification, version 4.0. online, Oracle Corporation, 2017.

[51] Mauro Cherubini, Gina Venolia, Rob DeLine, and Amy J. Ko. *Let's Go to the Whiteboard: How and Why Software Developers Use Drawings*, pages 557–566. Association for Computing Machinery, New York, NY, USA, 2007.

[52] Brian Chess and Jacob West. *Secure Programming with Static Analysis*. Addison-Wesley Professional, first edition, 2007.

[53] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, pages 151—158, New York, NY, USA, 1971. Association for Computing Machinery.

[54] Nigel Deakin. Java message service, version 2.0 revision a. online, Oracle Corporation, March 2015.

[55] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In Mario Tokoro and Remo Pareschi, editors, *ECOOP'95 — Object-Oriented Programming, 9th European Conference, Åarhus, Denmark, August 7–11, 1995*, pages 77–101, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.

[56] Tom DeMarco. *Structured analysis and system specification*. Yourdon computing series. Yourdon, Upper Saddle River, NJ, 1979.

[57] Linda DeMichiel and Lukas Jungmann. *JSR-000338 Java Persistence 2.2 Specification Maintenance Release*. Oracle America, Inc., July 2017.

[58] Linda DeMichiel and Bill Shannon. *JSR 000366 Java Platform, Enterprise Edition 8 Specification*. Oracle America, Inc., August 2017.

[59] Linda G. DeMichiel. Enterprise javabeans™ specification, version 2.1. online, Sun Microsystems, 2003.

[60] Linda G. DeMichiel and Michael Keith. Jsr 220: Enterprise javabeans™, version 3.0. online, Sun Microsystems, 2006.

[61] Jörg Desel and Gabriel Juhás. *"What Is a Petri Net?" Informal Answers for the Informed Reader*, pages 1–25. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.

[62] D. Dhillon and V. Mishra. Applied threat driven security verification. In *2018 IEEE Cybersecurity Development (SecDev)*, pages 135–135, Sep. 2018.

[63] Danny Dhillon. Developer-driven threat modeling: Lessons learned in the trenches. *IEEE Security and Privacy*, 9(4):41–47, 2011.

[64] S. Ducasse and D. Pollet. Software architecture reconstruction: A process-oriented taxonomy. *IEEE Transactions on Software Engineering*, 35(4):573–591, 2009.

[65] Claudia Eckert. *IT-Sicherheit: Konzepte – Verfahren – Protokolle*. Oldenbourg, München, 10., aktualisierte und korr. aufl. edition, 2018.

[66] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science. An EATCS Series)*. Springer-Verlag, Berlin, Heidelberg, 2006.

[67] T. Eisenbarth, R. Koschke, and G. Vogel. Static trace extraction. In *Proceedings of the 9th Working Conference on Reverse Engineering, 2002*, pages 128–137, 2002.

[68] Sascha Fahl, Marian Harbach, Thomas Muders, Lars Baumgärtner, Bernd Freisleben, and Matthew Smith. Why eve and mallory love android: An analysis of android ssl (in)security. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, page 50–61, New York, NY, USA, 2012. Association for Computing Machinery.

[69] Eduardo B. Fernandez. Security patterns and secure systems design. In *Proceedings of the 45th Annual Southeast Regional Conference*, ACM-SE 45, page 510, New York, NY, USA, 2007. Association for Computing Machinery.

[70] Eduardo Fernandez-Buglioni. *Security Patterns in Practice: Designing Secure Architectures Using Software Patterns*. Wiley Publishing, 1st edition, 2013.

[71] David Ferraiolo and Richard Kuhn. Role-based access control. In *15th NIST-NCSC National Computer Security Conference*, pages 554–563, 1992.

[72] Brian Fitzgerald and Klaas-Jan Stol. Continuous software engineering and beyond: Trends and challenges. In *Proceedings of the 1st International Workshop on Rapid Continuous Software Engineering*, RCoSE 2014, page 1–9, New York, NY, USA, 2014. Association for Computing Machinery.

[73] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. Cypher: An evolving query language for property graphs. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 1433–1445, New York, NY, USA, 2018. Association for Computing Machinery.

[74] Simone Frau, Roberto Gorrieri, and Carlo Ferigato. Petri net security checker: Structural non-interference at work. In Pierpaolo Degano, Joshua Guttman, and Fabio Martinelli, editors, *Formal Aspects in Security and Trust*, pages 210–225, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[75] Jakob Freund and Bernd Rücker. *Praxishandbuch BPMN 2.0*. Hanser, 3rd edition, 2012.

[76] Bundesamt für Sicherheit in der Informationstechnologie. Kryptographische verfahren: Empfehlungen und schlüssellängen. Technical report, Bundesamt für Sicherheit in der Informationstechnologie, 03 2021.

[77] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., USA, 1995.

[78] J. Garcia, I. Ivkovic, and N. Medvidovic. A comparative analysis of software architecture recovery techniques. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 486–496, 2013.

[79] David Garlan, Felix Bachmann, James Ivers, Judith Stafford, Len Bass, Paul Clements, and Paulo Merson. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley Professional, 2nd edition, 2010.

[80] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, Daniel Smith, and Gavin Bierman. *The Java Language Specification, Java SE 15 Edition*, August 2020.

[81] Shawn Hernan, Scott Lambert, Tomasz Ostwald, and Adam Shostack. Uncover security design flaws using the STRIDE approach. *MSDN Magazine*, November 2006.

[82] Christine Hofmeister, Robert Nord, and Dilip Soni. *Applied Software Architecture*. Addison-Wesley, Reading, MA, USA, 1st edition, 1999.

[83] Michael Howard and Steve Lipner. *The Security Development Lifecycle*. Microsoft Press, USA, 2006.

[84] Michael Howard and Steve Lipner. *The Security Development Lifecycle: SDL: A Process for Developing Demonstrably More Secure Software*. Microsoft Press, May 2006.

[85] G. Hurlburt. "good enough" security: The best we'll ever have. *Computer*, 49(7):98–101, 2016.

[86] ISO/IEC/IEEE. Systems and Software Engineering – Architecture Description. *ISO/IEC/IEEE 42010:2011(E) (Revision of ISO/IEC 42010:2007 and IEEE Std 1471-2000)*, pages 1–46, 1 2011.

[87] Eric Jendrock, Jennifer Ball, Debbie Carson, Ian Evans, Scott Fordin, and Kim Haase. The Java EE 5 Tutorial. online, Oracle Corporation, 2010. https://docs.oracle.com/javaee/5/tutorial/doc/.

[88] A. A. A. Jilani, A. Nadeem, T. Kim, and E. Cho. Formal representations of the data flow diagram: A survey. In *2008 Advanced Software Engineering and Its Applications*, pages 153–158, 2008.

[89] Christian Jung, Manuel Rudolph, and Reinhard Schwarz. Security evaluation of service-oriented systems using the sisoa method. *Int. J. Secur. Softw. Eng.*, 2(4):19–33, 2011.

[90] Jan Jürjens. Towards development of secure systems using umlsec. In Heinrich Hußmann, editor, *Fundamental Approaches to Software Engineering, 4th International Conference, FASE 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings*, volume 2029 of *Lecture Notes in Computer Science*, pages 187–200. Springer, 2001.

[91] Jan Jürjens. Umlsec: Extending uml for secure systems development. In Jean-Marc Jézéquel, Heinrich Hussmann, and Stephen Cook, editors, *UML 2002 — The Unified Modeling Language*, pages 412–425, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[92] Jan Jürjens. Model-based security testing using umlsec: A case study. *Electronic Notes in Theoretical Computer Science*, 220(1):93–104, 2008. Proceedings of the Fourth Workshop on Model Based Testing (MBT 2008).

[93] Jan Jürjens and Siv Hilde Houmb. Risk-driven development of security-critical systems using umlsec. In Ricardo Reis, editor, *Information Technology, Selected Tutorials, IFIP 18th World Computer Congress, Tutorials, 22-27 August 2004, Toulouse, France*, volume 157 of *IFIP*, pages 21–53. Kluwer/Springer, 2004.

[94] Jan Jürjens and Pasha Shabalin. Automated verification of umlsec models for security requirements. In Thomas Baar, Alfred Strohmeier, Ana M. D. Moreira, and Stephen J. Mellor, editors, *«UML» 2004 - The Unified Modelling Language: Modelling Languages and Applications. 7th International Conference, Lisbon, Portugal, October 11-15, 2004. Proceedings*, volume 3273 of *Lecture Notes in Computer Science*, pages 365–379. Springer, 2004.

[95] Kaarina Karppinen, Mikael Lindvall, and Lyly Yonkwa. Detecting security vulnerabilities with software architecture analysis tools. In *First International Conference on Software Testing Verification and Validation, ICST 2008, Lillehammer, Norway, April 9-11, 2008, Workshops Proceedings*, pages 262–268. IEEE Computer Society, 2008.

[96] Rick Kazman, Mark H. Klein, Mario Barbacci, Thomas A. Longstaff, Howard F. Lipson, and S. Jeromy Carrière. The architecture tradeoff analysis method. In *4th International Conference on Engineering of Complex Computer Systems (ICECCS '98), 10-14 August 1998, Monterey, CA, USA*, pages 68–78. IEEE Computer Society, 1998.

[97] Chris F. Kemerer and Mark C. Paulk. The impact of design and code reviews on software quality: An empirical study based on psp data. *IEEE Trans. Softw. Eng.*, 35(4):534–550, July 2009.

[98] Ebrahim Khalaj, Radu Vanciu, and Marwan Abi-Antoun. Is there value in reasoning about security at the architectural level: A comparative evaluation. In *Proceedings of the 2014 Symposium and Bootcamp on the Science of Security*, HotSoS '14, New York, NY, USA, 2014. Association for Computing Machinery.

[99] Gary A. Kildall. A unified approach to global program optimization. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '73, pages 194–206, New York, NY, USA, 1973. Association for Computing Machinery.

[100] Michael Kircher and Prashant Jain. Pooling. In Alan O'Callaghan, Jutta Eckstein, and Christa Schwanninger, editors, *Proceedings of the 7th European Conference on Pattern Languages of Programms (EuroPLoP '2002), Irsee, Germany, July 3-7, 2002*. UVK - Universitaetsverlag Konstanz, 2002.

[101] Rainer Koschke and Daniel Simon. Hierarchical reflexion models. In *Proceedings of the 10th Working Conference on Reverse Engineering*, WCRE '03, page 36, USA, 2003. IEEE Computer Society.

[102] Philippe Kruchten. The 4+1 View Model of Architecture. *IEEE Software*, 12(6):42–50, November 1995.

[103] D. Richard Kuhn, Edward J. Coyne, and Timothy R. Weil. Adding attributes to role-based access control. *Computer*, 43(6):79–81, 2010.

[104] Hannu Kuusela and Pallab Paul. A comparison of concurrent and retrospective verbal protocol analysis. *The American Journal of Psychology*, 113(3):387–404, 2000.

[105] Butler W. Lampson. Protection. *SIGOPS Oper. Syst. Rev.*, 8(1):18–24, January 1974.

[106] William Landi. Undecidability of static analysis. *ACM Transactions on Programming Languages and Systems*, 1(4):323–337, December 1992.

[107] Marc-André Laverdière, Bernhard J. Berger, and Ettore Merlo. Taint analysis of manual service compositions using cross-application call graphs. In Yann-Gaël Guéhéneuc, Bram Adams, and Alexander Serebrenik, editors, *22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015, Montreal, QC, Canada, March 2-6, 2015*, pages 585–589. IEEE Computer Society, 2015.

[108] H. F. Lawson, V. Kirova, and W. Rossak. A refinement of the ecbs architecture constituent. In *Proceedings of the 1995 International Symposium and Workshop on Systems Engineering of Computer-Based Systems*, pages 95–102, 1995.

[109] Ondřej Lhoták. Program analysis using binary decision diagrams. Technical report, School of Computer Science McGill University, Montreal, January 2006.

[110] Ondřej Lhoták and Laurie Hendren. Scaling java points-to analysis using spark. In *Proceedings of the 12th International Conference on Compiler Construction*, CC'03, pages 153–169, Berlin, Heidelberg, 2003. Springer-Verlag.

[111] Tim Lindholm, Frank Yellin, Gilad Bracha, Alex Buckley, and Daniel Smith. Jsr 392: The java® virtual machine specification, java se 17 edition. online, Oracle America, Inc., August 2021.

[112] Torsten Lodderstedt, David Basin, and Jürgen Doser. Secureuml: A uml-based modeling language for model-driven security. In Jean-Marc Jézéquel, Heinrich Hussmann, and Stephen Cook, editors, *UML 2002 — The Unified Modeling Language*, pages 426–441, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[113] Patrik Maier, Zhendong Ma, and Roderick Bloem. Towards a secure scrum process for agile web application development. In *Proceedings of the 12th International Conference on Availability, Reliability and Security*, ARES '17, New York, NY, USA, 2017. Association for Computing Machinery.

[114] Sönke Maseberg, Eric Bodden, Mehmet Kus, Achim Brucker, Siegfried Rasthofer, Bernhard Berger, Stephan Huber, Karsten Sohr, Paul Gerber, and Melanie Volkamer. Zertifizierte apps. In *Risiken kennen, Herausforderungen annehmen, Lösungen gestalten, Tagungsband zum 14. Deutscher*

IT-Sicherheitskongress des BSI 2015, pages 505–516. SecuMedia, January 2015.

[115] Gary McGraw. *Software Security: Building Security In*. Addison-Wesley Professional, 2006.

[116] Gail C. Murphy, David Notkin, and Kevin J. Sullivan. Software reflexion models: Bridging the gap between design and implementation. *IEEE Transaction on Software Engineering*, 27(4):364–380, April 2001.

[117] Anthony Nadalin, Chris Kaler, Ronald Monzillo, and Phillip Hallam-Bake. Web Services Security: SOAP Message Security 1.1. online, OASIS Open, February 2006. https://www.oasis-open.org/specs/index.php#wssv1.1.

[118] Rodrigue Wete Nguempnang, Bernhard J. Berger, and Karsten Sohr. enypd — entry points detector – jakarta server faces use case. In *2021 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2021. accepted for publication.

[119] Object Management Group. Business Process Model and Notation (OMG BPMN) – Version 2.0.2. Standard, Object Management Group (OMG), January 2014.

[120] Object Management Group. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification – Version 1.3. Standard, Object Management Group (OMG), June 2016.

[121] Object Management Group. OMG Unified Modeling Language (OMG UML) – Version 2.5.1. Standard, Object Management Group (OMG), December 2017.

[122] Object Management Group. OMG Systems Modeling Language (OMG SysML) – Version 1.6. Standard, Object Management Group (OMG), November 2019.

[123] Lars Patzina, Sven Patzina, Thorsten Piper, and Andy Schürr. Monitor petri nets for security monitoring. In *Proceedings of the International Workshop on Security and Dependability for Resource Constrained Embedded Systems*, S&D4RCES '10, New York, NY, USA, 2010. Association for Computing Machinery.

[124] Carl Adam Petri. *Kommunikation mit Automaten*. PhD thesis, Universität Hamburg, 1962.

[125] Olgierd Pieczul, Simon Foley, and Mary Ellen Zurko. Developer-centered security and the symmetry of ignorance. In *Proceedings of the 2017 New Security Paradigms Workshop*, NSPW 2017, page 46–56, New York, NY, USA, 2017. Association for Computing Machinery.

[126] Poonam Ponde, Shailaja Shirwaikar, and Christian Kreiner. An analytical study of security patterns. In *Proceedings of the 21st European Conference on Pattern Languages of Programs*, EuroPlop '16, New York, NY, USA, 2016. Association for Computing Machinery.

[127] G. Ramalingam. The undecidability of aliasing. *ACM Transactions on Programming Languages and Systems*, 16(5):1467—1471, September 1994.

[128] Aoun Raza, Gunther Vogel, and Erhard Plödereder. Bauhaus – a tool suite for program analysis and reverse engineering. In Luís Miguel Pinho and Michael González Harbour, editors, *Reliable Software Technologies – Ada-Europe 2006*, pages 71–82, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[129] Trygve Reenskaug. Model-view-controller. online, XEROX PARC, December 1979.

[130] S. Rehman and K. Mustafa. Research on software design level security vulnerabilities. *SIGSOFT Softw. Eng. Notes*, 34(6):1–5, December 2009.

[131] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, pages 49–61, New York, NY, USA, 1995. Association for Computing Machinery.

[132] Kalle Rindell, Jukka Ruohonen, and Sami Hyrynsalmi. Surveying secure software development practices in finland. In *Proceedings of the 13th International Conference on Availability, Reliability and Security*, ARES 2018, New York, NY, USA, 2018. Association for Computing Machinery.

[133] Marko A. Rodriguez. The gremlin graph traversal machine and language (invited talk). In *Proceedings of the 15th Symposium on Database Programming Languages*, DBPL 2015, pages 1—10, New York, NY, USA, 2015. Association for Computing Machinery.

[134] Mark Roth and Eduardo Pelegrî-Llopart. Javaserver pages™ specification, version 2.0. online, Oracle Corporation, November 2003.

[135] Nick Rozanski and Eóin Woods. *Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives*. Addison-Wesley Professional, 2nd edition, 2011.

[136] Thomas Ruhroth and Jan Jürjens. Supporting security assurance in the context of evolution: Modular modeling and analysis with umlsec. In *14th International IEEE Symposium on High-Assurance Systems Engineering, HASE 2012, Omaha, NE, USA, October 25-27, 2012*, pages 177–184. IEEE Computer Society, 2012.

[137] Kenneth Saks. Jsr 318: Enterprise javabeans™, version 3.1. online, Sun Microsystems, 2009.

[138] R. Sandhu. Good-enough security. *IEEE Internet Computing*, 7(1):66–68, 2003.

[139] Ravi S. Sandhu. Role-based access control. volume 46 of *Advances in Computers*, pages 237–286. Elsevier, 1998.

[140] K. Sartipi. Software architecture recovery based on pattern matching. In *International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings.*, pages 293–296, 2003.

[141] M Sharir and A Pnueli. *Two approaches to interprocedural data flow analysis*. New York Univ. Comput. Sci. Dept., New York, NY, 1978.

[142] Kelly Short. Security hacks and the lasting impact on retailers. Technical report, Interactions Consumer Experience Marketing, Inc, 4 2016.

[143] Adam Shostack. *Threat Modeling: Designing for Security*. Wiley Publishing, 1st edition, 2014.

[144] Laurens Sion, Pierre Dewitte, Dimitri Van Landuyt, Kim Wuyts, Ivo Emanuilov, Peggy Valcke, and Wouter Joosen. An architectural view for data protection by design. In *2019 IEEE International Conference on Software Architecture (ICSA)*, pages 11–20, 2019.

[145] Laurens Sion, Katja Tuma, Riccardo Scandariato, Koen Yskout, and Wouter Joosen. Towards automated security design flaw detection. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*, pages 49–56, 2019.

[146] Laurens Sion, Koen Yskout, Dimitri Van Landuyt, and Wouter Joosen. Risk-based design security analysis. In *Proceedings of the 1st International*

Workshop on Security Awareness from Design to Deployment, SEAD '18, page 11–18, New York, NY, USA, 2018. Association for Computing Machinery.

[147] Karsten Sohr and Bernhard J. Berger. Idea: Towards architecture-centric security analysis of software. In Fabio Massacci, Dan S. Wallach, and Nicola Zannone, editors, *Engineering Secure Software and Systems, Second International Symposium, ESSoS 2010, Pisa, Italy, February 3-4, 2010. Proceedings*, volume 5965 of *Lecture Notes in Computer Science*, pages 70–78. Springer, 2010.

[148] Fabio N. Souza, Roberto D. Arteiro, Nelson S. Rosa, and Paulo R. M. Maciel. Performance models for the instance pooling mechanism of the jboss application server. In *2008 IEEE International Performance, Computing and Communications Conference*, pages 135–143, 2008.

[149] William C. Stratton, Deane E. Sibol, Mikael Lindvall, and Patricia Costa. Technology infusion of save into the ground software development process for nasa missions at jhu/apl. In *2007 IEEE Aerospace Conference*, pages 1–15, March 2007.

[150] Harald Störrle and Jan Hendrik Hausmann. Towards a formal semantics of uml 2.0 activities. In Peter Liggesmeyer, Klaus Pohl, and Michael Goedicke, editors, *Software Engineering 2005*, pages 117–128, Bonn, 2005. Gesellschaft für Informatik e.V.

[151] S. Subashini and V. Kavitha. Review: A survey on security issues in service delivery models of cloud computing. *J. Netw. Comput. Appl.*, 34(1):1–11, January 2011.

[152] Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. Practical virtual method call resolution for java. *SIGPLAN Notices*, 35(10):264–280, October 2000.

[153] Frank Swiderski and Window Snyder. *Threat Modeling*. Microsoft Press, USA, 2004.

[154] Izar Tarandach and Matthew J. Coles. *Threat Modeling*. USA, November 2020.

[155] Jakarta EE Platform Team. *Jakarta EE Platform, Version 8*. Eclipse Foundation, August 2019.

[156] Inger Anne Tøndel, Daniela Soares Cruzes, and Martin Gilje Jaatun. Achieving "good enough" software security: The role of objectivity. In *Proceedings of the Evaluation and Assessment in Software Engineering*, EASE '20, page 360–365, New York, NY, USA, 2020. Association for Computing Machinery.

[157] K. Tuma, G. Calikli, and R. Scandariato. Threat analysis of software systems: A systematic literature review. *Journal of Systems and Software*, 144:275–294, 2018.

[158] Katja Tuma, Danial Hosseini, Kyriakos Malamas, and Riccardo Scandariato. Inspection guidelines to identify security design flaws. In *Proceedings of the 13th European Conference on Software Architecture - Volume 2*, ECSA '19, page 116–122, New York, NY, USA, 2019. Association for Computing Machinery.

[159] Katja Tuma, Riccardo Scandariato, Mathias Widman, and Christian Sandberg. Towards security threats that matter. In Sokratis K. Katsikas, Frédéric Cuppens, Nora Cuppens, Costas Lambrinoudakis, Christos Kalloniatis, John Mylopoulos, Annie Antón, and Stefanos Gritzalis, editors, *Computer Security*, pages 47–62, Cham, 2018. Springer International Publishing.

[160] Katja Tuma, Laurens Sion, Riccardo Scandariato, and Koen 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, page 332–342, New York, NY, USA, 2020. Association for Computing Machinery.

[161] Margus Välja, Fredrik Heiding, Ulrik Franke, and Robert Lagerström. Automating threat modeling using an ontology framework. *Cybersecur.*, 3(1):19, 2020.

[162] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '99, page 13. IBM Press, 1999.

[163] M. van Hilst and E. Fernández. Reverse engineering to detect security patterns in code. In *Proceedings of the 1st International Workshop on Software Patterns and Quality*, 2007.

[164] Radu Vanciu and Marwan Abi-Antoun. Ownership object graphs with dataflow edges. In *2012 19th Working Conference on Reverse Engineering*, pages 267–276, 2012.

[165] Dániel Varró, Gábor Bergmann, Ábel Hegedüs, Ákos Horváth, István Ráth, and Zoltán Ujhelyi. Road to a reactive and incremental model transformation platform: three generations of the viatra framework. *Software & Systems Modeling*, 15(3):609–629, 05/2016 2016.

[166] Margus Välja, Fredrik Heiding, Ulrik Franke, and Lagerström Robert. Automating threat modeling using an ontology framework. *Cybersecurity*, 3, 10 2020.

[167] W3C SPARQL Working Group. SPARQL 1.1. Technical report, March 2013.

[168] Claes Wohlin. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*, EASE '14, New York, NY, USA, 2014. Association for Computing Machinery.

[169] eXtensible Access Control Markup Language (XACML) Version 3.0. Standard, OASIS, January 2013.

[170] Xiao Xiao and Charles Zhang. Geometric encoding: Forging the high performance context sensitive points-to analysis for java. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA'11, page 188–198, New York, NY, USA, 2011. Association for Computing Machinery.

[171] Koen Yskout, Riccardo Scandariato, and Wouter Joosen. Do security patterns really help designers? In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, page 292–302. IEEE Press, 2015.

[172] Wangyang Yu, Zhijun Ding, Lu Liu, Xiaoming Wang, and Richard David Crossley. Petri net-based methods for analyzing structural security in e-commerce business processes. *Future Generation Computer Systems*, 109:611–620, 2020.

[173] M. Zahid, Z. Mehmmod, and I. Inayat. Evolution in software architecture recovery techniques — a survey. In *2017 13th International Conference on Emerging Technologies (ICET)*, pages 1–6, 2017.

[174] Peter D. Zegzhda, Dmitry P. Zegzhda, Maxim O. Kalinin, and Artem S. Konoplev. Security modeling of grid systems using petri nets. In Igor Kotenko and Victor Skormin, editors, *Computer Network Security*, pages 299–308, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[175] Henning Ziegler. Analyse der verwendung von kryptographie-apis in java-basierten anwendungen. Technical report, Universität Bremen, 2016.