Architectural Aspects of Software Security

Dr. Karsten Sohr

Cumulative Habilitation Thesis

Submitted to the Faculty of Mathematics and Computer Science at the
University of Bremen, Germany

29th January 2020

Date of the habilitation colloquium: _____

Acknowledgements

This work reflects years of experience that I gained in the field of software security. Many colleagues have contributed to this result, via fruitful discussions as well as direct collaboration within several research projects. In particular, I would like to thank Bernhard J. Berger, Henk Birkholz, Michaela Bunke, Michael Drouineaud, Prof. Dr. Stefan Edelkamp, Dr. Carsten Elfers, Kai T. Hillmann, Florian Junge, Prof. Dr. Thomas Kemmerich, Christian Liebig, and Dr. Tanveer Mustafa for helping me develop the area of information security at the University of Bremen. Without their support and their loyalty, I could have never achieved these results.

Further thanks go to Prof. Dr. Martin Gogolla and Prof. Dr. Rainer Malaka. Prof. Dr. Martin Gogolla always promoted my research and Prof. Dr. Rainer Malaka supported my work at the TZI, although no Chair of Information Security exists at the University of Bremen.

I'm also grateful to Prof. Dr. Rainer Koschke for being open for the software security topic and integrating it into the research agenda of the Software Engineering Group at the University of Bremen. From the international point of view, I'm very thankful to Prof. Dr. Gail-Joon Ahn, Arizona State University at Tempe. He always believed in my work and gave me valuable help to improve my research. Prof. Dr. Carsten Bormann also contributed to the results of this thesis by early pointing out the importance of the information security field at the University of Bremen, at a time when information security played a relatively subordinate role both in industry and research.

Another essential aspect which contributed to the results of this habilitation thesis is my work at the TZI. This work led to a practical view on software security, for example, gained within the "BremSec-Forum", which I have cofounded in 2004 jointly with Siemens AG and the German Association for Data Protection and Data Security (GDD), Bremen. The discussions with security and data protection experts from enterprises and governmental agencies greatly influenced my work and finally allowed me to define several successful research projects. Also the contacts to the German Federal Office for Information Security (BSI), most notably with Mr. Thomas Caspers, helped me to advance my work in the field of software security.

Furthermore, thanks go to the German Federal Ministry of Research and Education (BMBF), who have supported my research with various research grants. Without this funding, my research would have certainly stopped years ago.

Lastly, and most importantly, I would deeply thank my family, Maria, Hanna, and Greta, who have suffered from my hard work and the really insecure job situation.

Abstract

Due to the increasing introduction of new technologies, such as mobile devices, Internet of Things, automotive controllers, and medical devices, software security plays a more and more important role. In particular, new risks arise for companies and private users as well, which software vendors must address adequately. An important task within such a Security Development Lifecycle is architectural risk analysis, which aims to identify and mitigate weaknesses in the software design. Often software vendors, however, do not carry out this task as it requires profound expertise and hence is quite expensive. This may finally lead to higher risks for customers.

This habilitation thesis ("Habilitationsschrift") deals with the topic of how such architectural risk analysis can be carried our more systematically by using supporting tools. This work is based upon results that we gained from several funded R&D projects at the University of Bremen. In particular, we discuss how an organization can systematically define and validate role-based policies within the process of software design.

This habilitation thesis also introduces different approaches to semi-automatically reconstructing the implemented security architecture from the source code of Java-based applications. This reconstructed security architecture is then subject to security analyses. Our approaches can be utilized in the early software development phases. However, they can also be applied after the software has been implemented. We believe that our work will be the basis for a new class of tools, which software vendors can integrate into their Security Development Lifecycle to simplify the task of architectural risk analysis.

Zusammenfassung

Mit der zunehmenden Verbreitung neuer Technologien wie z.B. mobile Endgeräte, Internet of Things, Automotive oder Medizintechnik stellt sich vermehrt die Frage nach der Software-Sicherheit im Sinne des englischen Begriffes Security. Insbesondere entstehen für Unternehmen, aber auch den privaten Nutzer neue Risiken, die Software-Hersteller angemessen adressieren müssen. Ein wesentlicher Schritt eines solchen sicheren Entwicklungsprozesses (*Security Development Lifecycle*) von Software ist die architekturelle Risikoanalyse, bei der grundlegende Schwächen idealerweise bereits während des Designs der Software identifiziert und ggf. behoben werden. Oftmals wird dieser Schritt jedoch von Software-Herstellern nicht durchgeführt, da die architekturelle Risikoanalyse eine hohe Expertise erfordert und damit zu teuer ist, mit entsprechendem Risiko für die Nutzer der Software.

Die vorgelegte Habilitationsschrift befasst sich mit der Fragestellung, wie die architekturelle Risikoanalyse systematischer und vor allem werkzeuggestützt durchgeführt werden kann. Hierbei wird auf Ergebnisse verschiedener Forschungsprojekte, die über einen Zeitraum von mehr als einem Jahrzehnt an der Universität Bremen durchgeführt worden sind, zurückgegriffen. Insbesondere diskutieren wir, wie rollenbasierte Sicherheitsregeln (*role-based policies*) systematisch definiert und dokumentiert werden können.

Darüber hinaus behandelt die vorliegende Arbeit Ansätze, wie die in der Software bereits implementierte Sicherheitsarchitektur rekonstruiert und (semi-automatisch) analysiert werden kann. Diese Ansätze können in den frühen Entwicklungsphasen, aber auch nach Implementierung der Software eingesetzt werden. Wir sehen unsere Forschungsarbeit als Basis für eine neue Klasse von Werkzeugen, die Software-Hersteller in ihren sicheren Entwicklungsprozess integrieren können, um den Schritt der architekturellen Risikoanalyse zu vereinfachen.

Contents

Chapter 1
Introduction

Software is nearly ubiquitous these days. Not only does it run on classical IT systems such as PCs and servers, but also on e.g. smartphones, tablet PCs, industrial controllers, automotive controllers or medical devices—with the advent of the Internet of Things (IoT) this trend will certainly continue. Many of these applications are security-critical and hence security holes can have tremendous consequences in some cases. One classical example are the attacks on the nuclear power station Buschehr in Iran, which have been partly possible due to security holes in the software of an industrial controller [Langner, 2013]. In other cases, successful attacks on automotive controllers have been reported [Koscher et al., 2010; Miller and Valasek, 2015]. These attacks were possible because the entertainment system in a car was not adequately separated from the system controlling, for example, the brakes.

The importance of software security is highlighted by the fact that from year to year still many critical vulnerabilities in software, including operating systems, database systems and applications, are listed in the National Vulnerability Database (NVD), which is run by the National Institute of Standards and Technologies of the US (NIST). As shown in Figure 1, more than 4,000 high-severity vulnerabilities have been reported each year between 2017 and 2019, with a record high in 2019. This observation underlines the increasing relevance of software security.

Large software vendors, such as Microsoft, SAP, Google, and Adobe, have reacted on the software security problem by establishing their own security teams. What is more, they have introduced specific security processes, called Security Development Lifecycle (SDL), to follow a more systematic security approach during software development [International Organization for Standardization, 2011]. These processes mostly encompass different steps including security requirements engineering, architectural risk analysis, (tool-supported or manual) code reviews, security testing, and penetration testing.

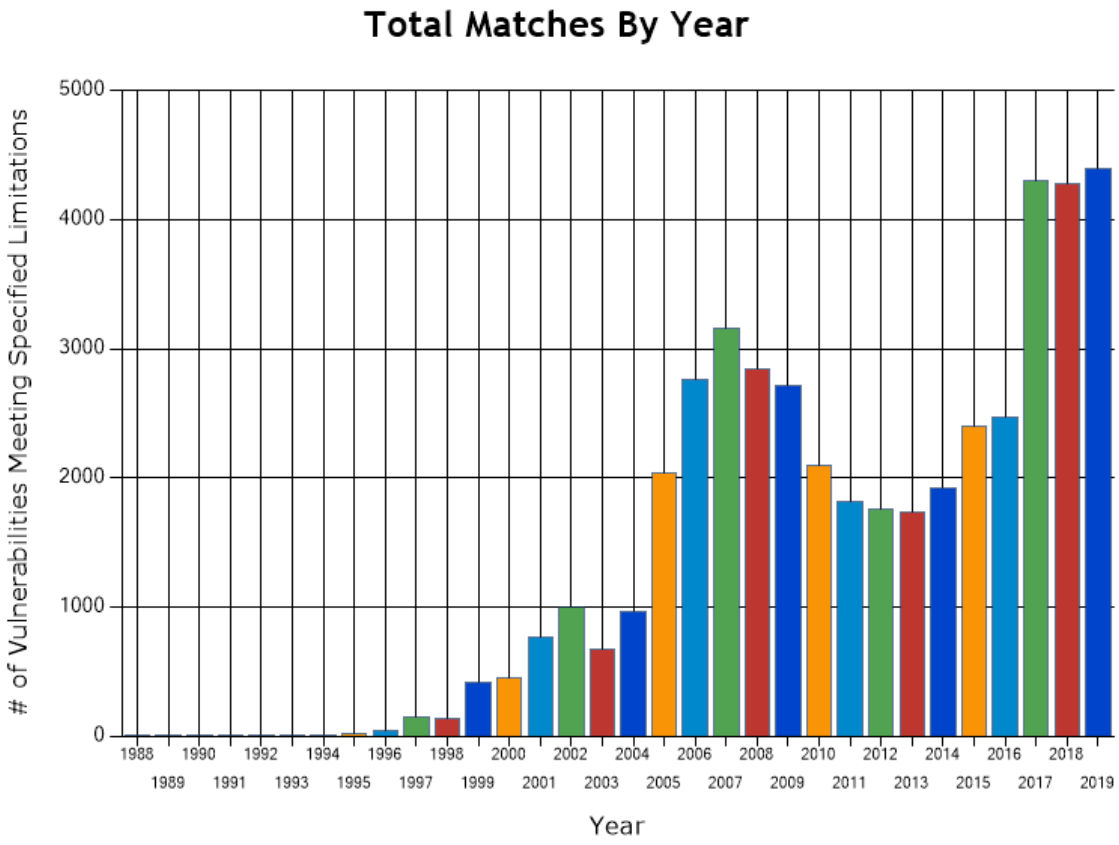

Figure 1: Development of high-severity vulnerabilities over the years; figure generated from the NVD web site[1] according to CVSS version 2.

While penetration tests of the deployed system and tool-supported code reviews, which aim to detect low-level programming bugs (such as buffer overflows or code injection vulnerabilities) at the source-code level, have been well-established, architectural risks analysis is often not applied to a sufficient degree [McGraw, 2013]. Architectural risk analysis, however, is an important subtask of an SDL because it attempts to detect foundational security defects in the software architecture, often called "flaws". Typical examples are missing encryption of confidential data on devices as well as on communications, missing or inconsistent rules for access control, over-privileged applications or insecure usage of software frameworks.

Software vendors tend to overlook architectural risk analysis for two reasons. First, one needs profound expertise to carry out this step successfully. It is not sufficient to be a network security expert for this task as one needs much experience on software development, software architectures and software frameworks. Being a software developer alone, however, also does not help

[1] http://web.nvd.nist.gov/view/vuln/statistics

here because one needs deep knowledge on security technologies, such as security protocols, access control systems or applied cryptography, which common developers and software architects usually do not have.

The second problem—somewhat related to the first one—lies in the manual nature of architectural risk analysis. The common practical process for this step, as for example defined by Microsoft's SDL [Microsoft, 2010] or the work by McGraw [McGraw, 2006], is to discuss the risks with the help of diagrams and come up with a security architecture that addresses the identified risks. Here, the analyst's expertise comes into play as she must have deep knowledge on typical risks that may exist in the software architecture under consideration. As this required expertise is scarce, the labor-intensive process of manual architectural risk analysis is too rarely carried out in practice [McGraw, 2013].

This habilitation thesis therefore deals with the research topic of how the process of architectural risk analysis can be systematized and automated. We take two perspectives with regard to this topic. First, we show how access control in applications can be implemented more systematically. In particular, we focus on role-based access control (RBAC), which is the de-facto standard for access control in organizations [Sandhu et al, 1996; American National Standards Institute, 2004]. We show how RBAC concepts can be formalized and use tools of practical relevance to specify and analyze role-based access control policies[2]. This step can be seen as a forward-engineering task, i.e., concrete role-based policies are derived from an organization's access control requirements. These policies can later be translated to access control features of specific target systems, e.g., SAP roles or roles for identity and access management systems, such as IBM Tivoli Access Manager.

The second perspective taken in this thesis is the reverse engineering point of view, i.e., we consider a backward engineering task. Given an implementation—we concentrate on Java-based applications for the sake of simplicity—we aim to reconstruct the security architecture from the program code semi-automatically, in the following called "implemented security architecture". We achieve this goal by utilizing static program analysis techniques including program slicing [Krinke, 2003]. This reconstructed security architecture can then be analyzed against common architectural rules as listed in the Common Weakness Enumeration (CWE)

[2] In the course of this thesis, we will use the more compact term "role-based policies" instead of "role-based access control policies".

and Common Attack Pattern Enumeration and Classification (CAPEC) published by MITRE [MITRE, 2014; MITRE, 2015] to identify architectural weaknesses. In addition, the security architecture can be visualized, which simplifies the process of understanding it including the identification of possible weaknesses. Beyond the visualization, we also sketch an approach that automatically infers constraints/invariants from program code.

We truly believe that both the forward and reverse engineering steps are valuable for simplifying architectural risk analysis within the frameworks of a Security Development Lifecycle. We also hope that our results may be useful for security architects or analysts and finally lead to more secure software in the future.

1.1 Structure of this Thesis

The remainder of this thesis is structured as follows. Chapter 2 discusses concepts of software security in general, with a focus on architectural risk analysis. In particular, we report on our experience that we gained from reporting vulnerabilities to security teams of several software vendors. In Chapter 3, we introduce two approaches to architectural risk analysis based on reverse engineering techniques. We discuss both approaches in the context of the smartphone platform Android as well as Java-based business applications. Chapter 4 introduces our approach to the specification and validation of role-based policies, a forward-engineering task. We use the Unified Modeling Language (UML) in conjunction with its Object Constraint Language (OCL) for this purpose. A kind of synthesis of both aspects is the topic of Chapter 5, where we present a technique of reverse-engineering the role-based policies implemented in a real-world application of a large enterprise. Thereafter, the reconstructed role-based policy can be validated with common UML validation tools. Finally, we conclude this thesis in Chapter 6 and give an outlook on future work.

1.2 How to Read this Thesis

This document is written as a cumulative habilitation thesis, i.e., it is composed of different conference and journal publications. At the end of Chapter 3, 4 and 5, we briefly discuss how those papers fit to the chapter in question. Most of the papers are collaborative work carried out with PhD students at the University of Bremen, most notably, Bernhard J. Berger, Mirco Kuhlmann, Dr. Tanveer Mustafa, Michael Drouineaud, and Michaela Bunke. These collaborations show that topics from the field of information security can be successfully addressed by

involving experts from other research fields of Computer Science including modeling IT and software systems, software technology, formal methods, and computer networks—this specific kind of collaboration is well-supported in the environment of the Center for Computing and Communications Technologies at the University of Bremen (TZI), Germany.

Chapter 2
Architectural Risk Analysis as a Key Task of a Security Development Lifecycle

In this chapter, we point out the importance of software security, in particular, the task of architectural risk analysis. Before discussing software security, we briefly explain the basic terminology of information security to provide the reader with the necessary basics in this field.

In the context of software security, we explain the Security Development Lifecycle (SDL) in more detail as it provides a systematic way for large software vendors to deal with software security in development projects. We highlight the aspect of architectural risk analysis and stress the fact that this step belongs to the most difficult and expert-driven tasks within an SDL. We illustrate this point with the help of typical real-world problems we encountered in various R&D projects, which we have carried out over the last years.

2.1 Information Security

Information security usually deals with the protection of IT systems and data from an attacker. This term is often differentiated from safety, which focuses on preventing malfunctioning inherent in the system itself [Eckert, 2013]. Various security objectives may have to be achieved by an IT system, including the following (see also the book by Anderson for a more exhaustive discussion [Anderson, 2008]):

- **Confidentiality**: Read access to sensitive data (e.g., user credentials, credit card numbers, and financial reports) must be constrained from unauthorized users of an IT system. Confidentiality is often achieved by employing cryptography.
- **Integrity**: Data must not be manipulated in an unauthorized way. Cryptographic mechanisms, such as cryptographic hash functions or digital signatures, are often used to implement integrity requirements.
- **Availability**: Access to IT systems and services must be guaranteed. For example, in an industrial setting, production may not stand still for a longer period of time; for a Web-based e-commerce application, the web server must always be accessible.

- **Authenticity**: Data must be integer, fresh (e.g., not being a copy) and their origin must be authenticated.
- **Non-repudiation**: Certain actions may not be repudiated by communication partners. In case of a dispute, the action must be proven to a trusted third-party.

For enforcing security objectives often the terms authentication, authorization, and audit are of interest (AAA). **Authentication** refers to the identification of certain subjects/principals accessing an IT system. **Authorization** or access control ensures that only subjects may access data if they have been allowed to do so. **Audit** means that certain activities are protocolled, e.g., in a log file. Auditing, for example, allows a security administrator to log security-relevant information, which can later be used for forensic purposes.

To build secure systems that must satisfy one or more of the aforementioned security objectives, design principles for secure systems should be followed. In their seminal work Saltzer and Schroeder formulated eight design principles, which have been shown important to realize secure systems [Saltzer and Schroeder, 1975]. Of them, one well-known principle is the "Principle of Least Privilege", which means that applications/systems should only run with the minimum access rights necessary to achieve their purposes. Another more and more relevant principle is "Psychological Acceptability", which states that IT security mechanism must be developed in a usable way; otherwise users will simply turn off or even ignore security completely.

To implement systems that satisfy security objectives in a reasonable way, the underlying software—be it system or application software—must be developed systematically with security in mind, during all development phases. The following sections discuss software security in more detail, demonstrating that it is a subfield of information security of increasing importance.

2.2 Software Security

End of the 90s of the last century, software security was more or less achieved by a penetrate-and-patch approach [Anderson, 2008]. Large software vendors mostly reacted on reported vulnerabilities and patched them as soon as possible. No real software security process existed at that time at most software vendors (maybe, with the exception of small niches of highly security-critical systems, such as banking applications). With the increasing number of Internet-based attacks on IT systems ("worms") at the beginning of the 2000s, large software vendors changed their policy regarding software security. It became clear that more systematic ap-

proaches to software security were necessary and that development processes had to be adequately adjusted to security needs. According to the software security expert McGraw, three trends contribute to the high number of security problems, which can be found in software: the growing complexity of software (e.g., MS Windows XP consisted of about 40 million lines of code), the increasing connectivity of software (e.g., formerly internal legacy software is exposed to the outside via an http interface), and extensibility of software systems (e.g., a browser can be extended by plugins or a smartphone can be extended by mobile applications). McGraw further coined the term "Trinity of trouble" to express the problems caused by these three trends [McGraw, 2006].

Having these problems in mind, Microsoft introduced a Security Development Lifecycle (MS SDL) within the frameworks of the "Trustworthy Computing Initiative" [McGraw, 2006; Howard and Lipner, 2006]. This SDL served as a blueprint for the establishment of SDLs in other large enterprises as well, such as SAP SE, Adobe Inc., or Siemens AG. The process at SAP SE, for example, is described in a document, which shows steps similar to the Microsoft SDL [SAP, 2019]. Due to the relevance of MS SDL, we describe this approach to software security in more detail now.

2.2.1 The SDL from Microsoft

We now introduce the core concepts of Microsoft's SDL, but for the sake of brevity, we only give a simplified version here—the basic principles remain the same for the complete description. The MS SDL can be integrated with software development processes, and in fact, it does not presume a specific development model, such as the waterfall model or iterative development models. In principle, it can also be integrated with agile models, such as SCRUM.

The upper part of Figure 2 shows typical activities in a development process of software, including requirements engineering, design, verification (testing) and release. The lower part then depicts activities specific to security.

We explain a few activities exemplarily to give the reader a feeling of the ideas behind this process. One should bear in mind that this process is carried out as a cycle, i.e. the single steps are repeated, although the figure does not directly show this aspect.

One mostly neglected activity is security training. Many security problems manifesting in Android applications, for example, stem from wrongly implemented cryptography [Egele et al., 2013] or communications security (problems in the implementation of TLS clients) [Fahl et

al., 2012]. As development teams rather than central security teams are responsible for developing secure code, it is an important task to educate developers (at least one member per team) concerning security.

Requirements engineering can also be adjusted to security aspects. For example, in a clinical information system, one might demand specific security requirements, such as "electronic patient records can only be read and written by physicians belonging to the same ward as the patient". Another security requirement in this application context is the condition that patient data must be encrypted and protected against manipulation by means of digital signatures.

A further activity is the security analysis of the software design (**security-by-design** approach). Subactivities include the analysis of the application's attack surface (roughly speaking, a security analysis of all entry and exit points of the software) and Threat Modeling. The latter point will be described in more detail in Section 2.3.2 as its concepts will be used in our reverse-engineering approach.

The implementation itself can also be subject to security analyses. Specifically, the program code can be checked with the help of a static source code analyzer to detect low-level programming bugs, such as buffer overflows or code injection vulnerabilities [Chess and West, 2007]. Typical static code analyzers include HP Fortify SCA [Hewlett-Packard, 2016], Coverity Prevent [Synopsis, 2016], and IBM AppScan [IBM, 2016].

In addition to static testing, dynamic analysis and Fuzz testing can be carried out against the running application. These steps are usually part of the "verification" phase within the SDL. Fuzz testing, which is often used for testing web applications, automatically fires test packets against the running application to identify security bugs, e.g., SQL injection or Cross-Site Scripting vulnerabilities. Often, Fuzz testing is also a part of an activity called "penetration testing", where a testing team (sometimes externally hired) tries to attack the application in operation. Furthermore, the attack surface determined in the design phase can be utilized to streamline dynamic testing against known external interfaces.

Further activities of the MS SDL concern the release and response phases. Given that complex software systems will contain security bugs, a response plan is necessary in case security holes are reported (e.g., through external reviewers, researchers or consultants). Large software ven-

dors have established central Computer Emergency Response Teams (CERTs), who are responsible for the communication with external security testers in case of a reported bug. Communication with the development teams is usually only indirectly possible via these CERTs.

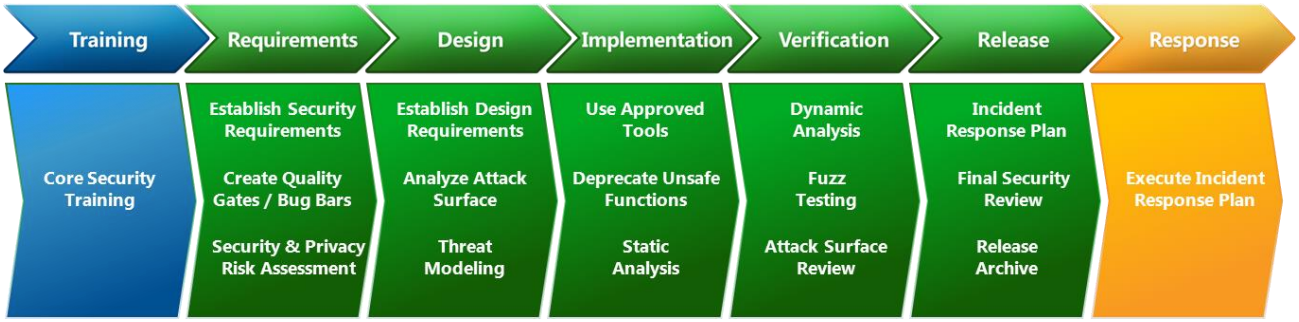

Figure 2: A simplified version of Microsoft's SDL (figure taken from [Microsoft, 2010]).

2.2.2 The Seven Touchpoints from McGraw

McGraw describes an SDL similar to the MS SDL, but refines it with respect to several aspects [McGraw, 2006]. Figure 3 depicts this process. The boxes in the lower part show artefacts that are produced within the software development lifecycle (e.g., architecture and design, code, test plans). In the upper part of the figure, McGraw shows the security-related activities that are injected into the software development process. The fact that the process is run several times is expressed by the large grey arrows. Each main security activity is called "touchpoint".

Most touchpoints have counterparts in Microsoft's SDL, e.g. the code review (tools) touchpoint correlates to the "static analysis" and "use tools" activities. Risk analysis is similar to Threat Modeling, but refines it in some points as we will discuss in Section 2.3.3. Penetration testing has some counterparts in the dynamic testing activities in MS SDL. Also, the secure operation touchpoint covers aspects of the "Execute Response Plan" activity.

One interesting touchpoint, not directly defined in MS SDL, are **abuse cases**. This term has been derived from the term "use case", but with the opposite meaning [Sindre and Opdahl, 2005]. Here, an analyst takes the perspective of an attacker and tries to find cases where the system under analysis can be attacked. One part of this activity is to define corner cases, e.g., defining negative values for the amount of a loan in a banking application or using a value that is out of the expected range. The motivation behind this step is that only considering positive security requirements may lead to an incomplete picture of an application's attack vectors.

Despite the differences in the detail, both approaches to software security follow a similar idea by injecting security activities into development processes. The introduction of such systematic

processes is a critical prerequisite to improving the security level of software. Both approaches claim to be independent of the concrete software development model as indicated above.

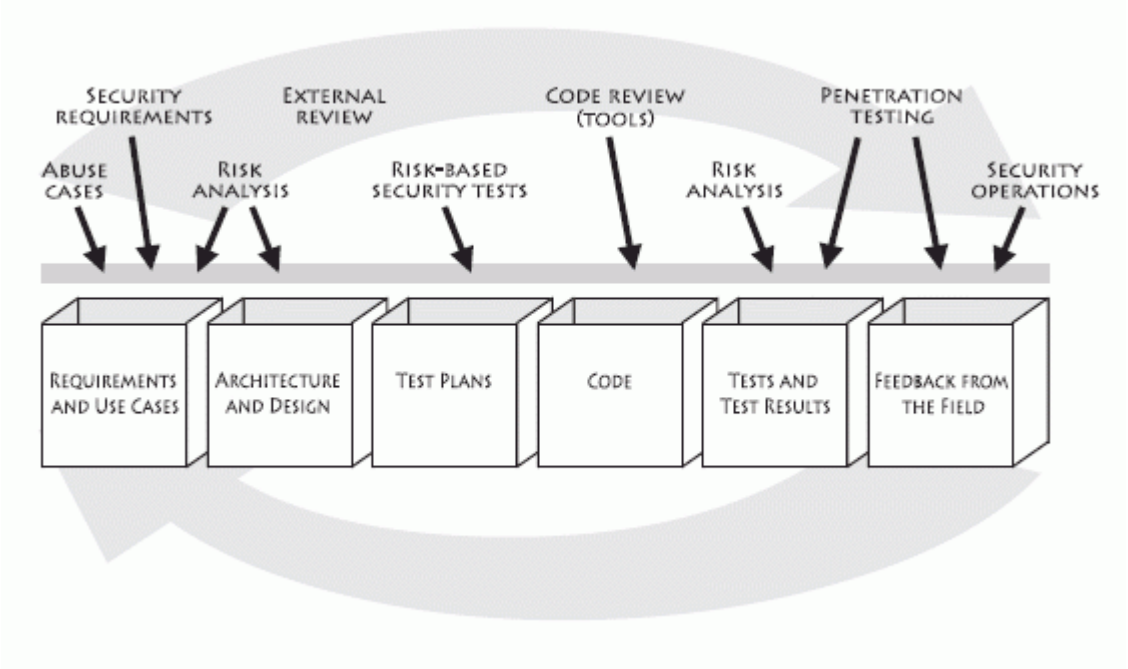

Figure 3: The Seven Touchpoints as defined by McGraw (figure taken from [McGraw, 2006]).

2.3 Architectural Risk Analysis

We first motivate the need of architectural risk analysis as an important step within an SDL by reporting on our own experience gained from several research projects. Thereafter, we describe Microsoft's Threat Modeling and architectural risk analysis according to McGraw. Architectural risk analysis can be seen as an important task implementing security-by-design approaches.

2.3.1 Motivating Examples from Practice

Within the frameworks of several research projects—including ZertApps [DLR, 2010] and ASKS [VDIVDE-IT, 2014], funded by the German Federal Ministry of Education and Research (BMBF)—we analyzed several software systems with respect to security. The analyzed set of software includes an e-government system, a logistics application, and a mobile app-controlled alarm system as well as several Android applications from large software vendors. In the following, we list some of our findings focussing on architectural aspects.

Privilege escalation in Sony Ericsson mobile phones: We begin this discussion with a flaw, which Adrian Nowak found in Sony Ericsson mobile phones in autumn 2007 [Sohr et al., 2011]. The flaw allowed an attacker to access the mobile phone's internal file system by using symbolic links. The mobile phone OS used the suffix .@ for symbolic links. Usually, file access APIs had to check that no file names are provided with this suffix. In an oversight, however, Sony Ericsson did not check this rule for the `rename` API, which allowed one to rename files. By employing the link concept, an attacker was able to overwrite system files such as root certificates and then to obtain manufacturer privileges on the phone. The security hole existed in about 70 million mobile devices, which could not be patched as no practically usable patch mechanism was available at that time. This flaw laid the starting point for our work in the area of software security.

Missing authorization in a Web-based logistics application: In a security-critical Web-based logistics application, access control checks have been implemented in JavaScript on the client-side. The problem is that an attacker can bypass these checks completely by writing her own client, which communicates directly with the server, i.e., the application was susceptible to a "Make the client invisible" attack as discussed in the CAPEC enumeration under the entry CAPEC-22 [MITRE, 2015]. The vendor should have provided access control checks on the server side.

Missing connection between authentication and authorization information in a logistics application: A Cloud-based logistics application did not use authentication information (e.g., login and password information) for authorization decisions appropriately. An attacker, being a legitimate user of the system, could then send messages into the system masquerading as a different user. As the logistics system can be regarded as a critical infrastructure, an attacker could provoke considerable monetary damage this way.

The logistics application consisted of two different parts, the first one implementing authentication mechanisms, whereas the second one performed the authorization checks. The problem was that both systems assumed that the other one had been responsible for connecting authentication and authorization information. This flaw confirms a statement by Anderson saying that often security problems arise at the boundaries between different systems [Anderson, 2008].

Incomplete authorization in an e-government application: This Java application showed a problem with respect to its authorization concept. The purpose of this e-government application

is to allow one to remotely sign documents. The signature is generated with the help of a smartcard, which contains the private signing key. The digital signature is considered legally binding.

While the application prevented an outside attacker from unauthorized access to the smartcard, an insider could sign documents of another system participant, e.g., of a different employee. This flaw passed Common Criteria certification [Common Criteria, 2009], although the Security Target (the document that contains the security requirements against which the IT product is checked during Common Criteria evaluation) did contain a corresponding security requirement [Berger et al., 2016].

Insecure random number generation in a Web-controlled alarm system: An alarm system, which can be remotely controlled by a Web or mobile application, contained an erroneous random number generation process [Hillmann, 2015]. If an attacker put the alarm system under stress by sending lots of the same IP packets, the entropy of the random values was significantly reduced. This flaw allowed for replay attacks without knowing the AES key that was used to secure the communications channel. The problem was that information from received IP packets was used as the entropy for generating the random values. As an attacker could send the same packets many times, the entropy was significantly reduced. Consequently, the random number generation process was under the control of an attacker—a classical architectural defect.

Insecure TLS connections in the app-controlled alarm system SPC Anywhere from Siemens (now Vanderbilt Industries): The app-controlled alarm system SPC Anywhere[3] from Siemens AG (now Vanderbilt Industries) contained a serious security hole concerning TLS security [Hillmann, 2015; ICS-CERT, 2015]. In particular, the decompiled Android version of the app contained the following code:

```
public void onReceivedSslError(WebView paramWebView, SslErrorHandler
    paramSslErrorHandler, SslError paramSslError)
  {
    paramSslErrorHandler.proceed();
  }
```

[3] https://play.google.com/store/apps/details?id=com.WebDevs.SPCanywhere&hl=de

Although the problem looks like a simple programming error at first glance—the TLS/SSL errors reported by the Android platform are simply ignored ("proceed") —, it shows a more fundamental security problem of the alarm system itself. The alarm system provided dynamic IP addresses (specifically, for private customers), which were used in the X.509 certificates for the authentication of the web server that was part of the alarm system. As the IP addresses were dynamic, X.509 authentication certificates had to be generated on the fly whenever the IP address was changed. To simplify this process, Siemens used *self-signed* certificates. This procedure broke the TLS/SSL authentication process during the handshake, and consequently the Android operating system had to report an error. Due to the `proceed()` call, however, the end user could never see any security warning, i.e., she had to assume that the connection between the mobile app and the alarm system was secure, which obviously was not the case.

We reported the flaw to Siemens AG and they managed to fix the problem after 15 months [Hillmann, 2015; ICS-CERT, 2015]. They came up with a completely revised architecture, now being implemented as a central portal system. The end user only indirectly communicates with the alarm system via a Web portal, which is made available by the Siemens AG; the server certificate of the portal is now valid[4]. However, the portal server can now be seen as a single point of failure, i.e., if an attacker can control it, she can control all the alarm systems that are connected to the portal. Furthermore, one cannot be sure that the connections between the portal and the alarm system have been adequately secured.

One general note should be given on TLS implementation flaws in Android applications. Vendors seem to argue that the attack surface of such TLS problems is relatively low because an attacker needs a "privileged network position" to perform a middleperson attack [ICS-CERT, 2015]. This assumption, however, has been proven wrong as the lately reported WiFi-security flaw [Vanhoef and Piessens, 2017]—which in itself is a considerable architectural security flaw—demonstrates: An attacker can easily obtain such a privileged network position via the (local) WiFi connection. This local attack scenario in particular is realistic for app-controlled IoT systems. An intruder, for example, can easily start her attack from the neighborhood of the victim's premises and then may turn off the alarm system.

[4] To be more precise, the current version of the SPC Anywhere app is still vulnerable. A new app-controlled alarm system, SPC Connect, including the fix of the alarm system itself has been published. This more secure Android application can be found under https://play.google.com/store/apps/details?id=com.siemens.spcconnect.

Problems with multi-threading in the industrial-controller app SIMATIC WinCC Sm@rtClient from Siemens: The Siemens SIMATIC WinCC Sm@rtClient app[5], which allows one to remotely access industrial controllers, contains a serious flaw related to multi-threading. The app usually presents the user a warning whenever a new X.509 certificate is used for communication. This step aims at preventing middleperson attacks on the SSL/TLS connection. The problem with this approach is that the check is done in a separate UI thread, i.e., the app silently sends the password via the main thread to the controller over a non-encrypted channel, regardless of the user's decision on the certificate. An attacker can then easily harvest the unencrypted password and directly access the industrial controller [Glander, 2017]. It seems that the developers/architects of this security-critical app have not thought about multi-threading security problems while implementing the app. This as well as another flaw have recently been confirmed by the Siemens Product CERT [Glander, 2017].

Insecure certificate check in the Siemens Siveillance VMS apps for Android and iOS: The Siemens Siveillance VMS app[6], an app for remotely controlling IP cameras, checks whether an SSL/TLS web server certificate has been changed. The following code implements this check[7]:

```
private void m7571a(X509Certificate[] x509CertificateArr) {
if (this.f4763a == 0) {
      m7572b(x509CertificateArr);
} else if (x509CertificateArr[0].hashCode() != this.f4763a) {
         C0151m.m7937a("TrustManager", "Certificate has changed !!!");
         m7572b(x509CertificateArr);
}
```

This code ought to prevent an attacker from presenting his own certificate and performing a man-in-the-middle attack (MITM) on the communication between the smartphone and the camera. In this MITM scenario, the attacker positions himself between the smartphone and the camera and masquerades as each partner. The code above only uses the Java hash code of the certificate Java object (`x509CertificateArr[0]`) for the comparison with the certificate that has already been stored in the app. Rather, it should have used a cryptographic hash or the Java `equals()` method to implement this "certificate pinning approach" correctly. An attacker can now easily generate his own certificate with the same hash code value as the stored

[5] https://play.google.com/store/apps/details?id=com.siemens.smartclient

[6] https://play.google.com/store/apps/details?id=com.siemens.siveillancevms

[7] This app code has been obfuscated by the vendor to impede readability for an attacker.

certificate, and hence an MITM attack is possible. The Siemens AG has confirmed the security hole and published a security advisory [Siemens ProductCERT, 2018].

Excessive Android permissions in an information app from the Telekom AG: The Android version of the Telekom.com app[8] published by the Telekom AG requested more than 20 permissions, although it only displayed harmless enterprise information of the Telekom AG. The permissions included writing and reading contacts, receiving SMS, accessing location information, accessing the camera, and recording audio. This app obviously violated the design principle of least privilege.

Exposing sensitive documents to other installed Android applications in the SAP Mobile Documents app: The SAP Mobile documents app[9] allows business customers to securely store and view business-critical documents on Android smartphones; the documents are fetched from a remote enterprise repository. This app exported temporarily security-critical documents to all other apps installed on the phone. The idea behind this behavior was to give viewer apps, such as the Acrobat reader app or the Excel app, access to these documents. However, the developers ignored the URI permission concept of Android [Enck et al., 2009], which was deliberately introduced by Google to handle such cases without any security violations. SAP SE finally published a security note on this (and another) flaw [SAP, 2015].

Android-service spoofing in SAP apps: Some SAP Android applications allow an attacking app to hijack SAP's central app-login service ("Client Hub") [Müller, 2015]. This central login service is implemented as a separate Android application. The Client Hub app, being part of the SAP Mobile SDK, helps one share common credentials between several enterprise apps, which is supposed to increase user experience. SAP users, however, are not required to use or even install this feature. An attacker can now implement an attack app with the same name as the Client Hub app (and with the same service name `com.sap.mobile.clientHub.MCIMService`) and try to get it installed on a device *before* the Client Hub app is actually installed. Then the other SAP apps assume that the Client Hub is available and inadvertently send credentials to the attacking app, which in turn can forward them to an attacker. The problem is here that the calling SAP enterprise apps do not verify whether the Client Hub app has been signed with the same developer certificate as the other SAP enterprise

[8] https://play.google.com/store/apps/details?id=com.telekom.www

[9] https://play.google.com/store/apps/details?id=com.sap.mcm.android

apps, i.e., whether the Client Hub app actually comes from SAP. In the context of a bachelor thesis, we implemented a proof-of-concept exploit, which can harvest credentials for the SAP IT Incident Management app for Android.[10] Since the Client Hub app belongs to the SAP Mobile SDK, similar problems may show up in other SAP Android apps, but we have not further investigated this point—at least, the SAP Fiori Client app[11] (more than 100,000 downloads) also contains such code. Lately, SAP SE has confirmed the vulnerability and provided a fix.

The aforementioned security flaws clearly demonstrate that even software from vendors, who have established an SDL and a product CERT (Siemens AG, SAP SE, Telekom AG) or who provide security-critical software (app-controlled alarm system, Common Criteria-certified e-government application), contains architectural security flaws. This observation confirms McGraw's statement that architectural risk analysis belongs to the most difficult, but also important tasks within the SDL [McGraw, 2013]. Our discussions with large vendors further support this statement by mentioning that central CERTs can only deal with the tip of the iceberg. A real advance would be to involve software architects of the developer teams in this task, but this seems to be difficult to achieve without any further support, e.g., education and tools.

2.3.2 Threat Modeling

Threat Modeling has been introduced by Microsoft within their SDL to identify security flaws early in the design stage [Shostack, 2014]. The idea is quite intuitive: The software architecture is represented in form of diagrams and possible risks (called "threats" in the terms of Microsoft) are identified and discussed with the help of these diagrams. As an appropriate representation form for such diagrams, Microsoft recommends dataflow diagrams (DFDs); other security experts, such as Bellovin, seem to favor DFD-like architectural representations as well [Bellovin, 2016]:

I approach system security analysis by boxes and arrows—a directed graph (not a tree!)—that shows input dependencies. (…) Also, see the "data flow diagrams" in [Shostack, 2014].

Figure 4 shows an example of a DFD for a Web-based client-server application. A DFD consists of processes (circles/ellipses), data flows (arrows), data stores (two parallel lines with a label between), external systems (rectangles), and trust boundaries (dashed lines) [Shostack,

[10] https://play.google.com/store/apps/details?id=com.sap.solman.activities

[11] https://play.google.com/store/apps/details?id=com.sap.fiori.client

2014; Hernan et al., 2006]. From the security viewpoint, trust boundaries are an important means to represent areas of different trust, e.g., a client is located in a different trust area than the server. Security measures should be established at those points of an application. In Figure 4, two trust boundaries are shown, one separates the client from the server, and the other lies between the server and the data base server.

Threat Modeling systematizes the process of finding risks in the software design, which is represented as DFDs. Microsoft proposes to assess all the elements of the DFD with respect to the violation of the aforementioned protection goals confidentiality, integrity, availability, non-repudiation, authorization, and authentication. For example, the connections between the client and the server in Figure 4 must be analyzed with respect to all the aforementioned protection-goal violations. Since this process might become tedious in many cases, later Dhillon proposed to focus only on central elements of the DFD, e.g., the connections crossing the trust boundaries in Figure 4 [Dhillon, 2011]. He further suggested to annotate the DFD with security measures/controls that have already been implemented, e.g., using secure channels for the connections in Figure 4. In the course of this thesis, we will follow these refining concepts because we pursue a reverse-engineering approach to architectural risk analysis and extract already implemented security mechanisms from program code.

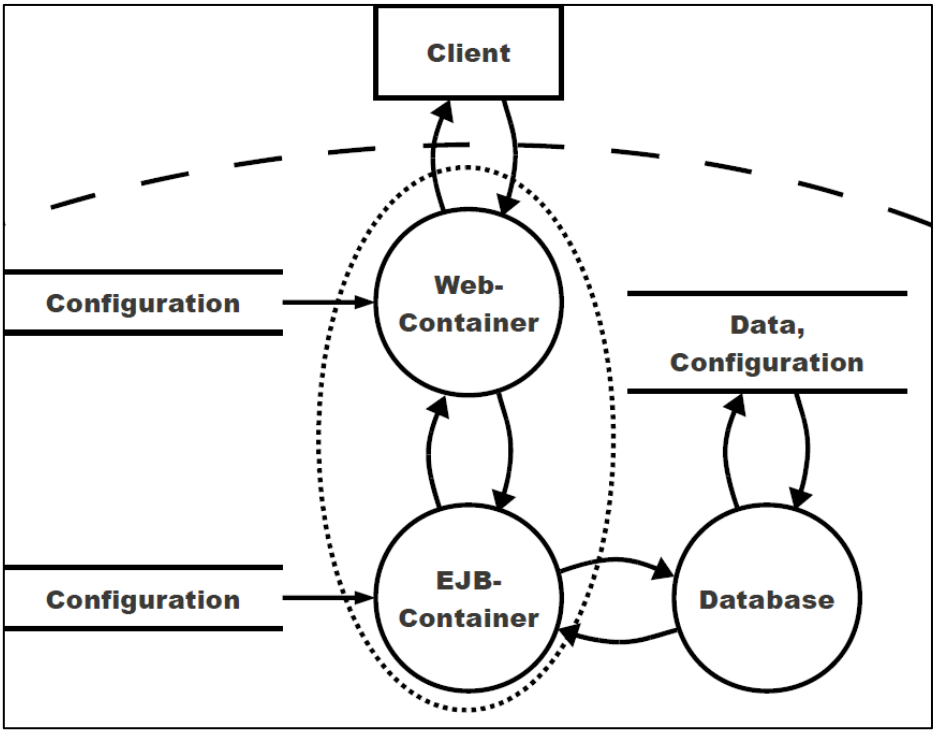

Figure 4: An example DFD for a Web-based business application (taken from [Berger et al., 2013]).

2.3.3 Architectural Risk Analysis according to McGraw

The approach introduced by McGraw encompasses the following steps [McGraw, 2006]:

1. Representation of the software design by means of diagrams: According to McGraw a forest-level overview is important to understand the security risks of an application. McGraw does not presume a certain diagram type, i.e., DFDs, UML diagrams or hand-written diagrams may suffice for this purpose.

2. Known-attack analysis: As mentioned earlier in this thesis, common lists exist that describe systematically security flaws in software, e.g., CWE [MITRE, 2014] and CAPEC [MITRE, 2015]. The IEEE Center for Secure Design has also published a document with the ten topmost security defects manifesting in software design [Arce et al., 2014].

   This step applies common lists of security flaws against the software design and attempts to identify corresponding risks. We later show how our approach to architectural risk analysis can be applied to identify known risks in software design (as listed in CWE).

3. System-specific attack analysis: This step tries to identify application-specific risks. It presumes profound software security expertise of the analyst to be effective. Usually, this step is performed in workshops together with software developers and architects. The security analyst tries to identify risks that, for example, arise through contradictions and ambiguities in the discussions during the workshop. In contrast to step 2, ambiguity analysis can hardly be automated.

4. Dependency analysis: Application software is often built upon many other components, such as software frameworks, middleware, and operating systems. Architectural risks must accommodate risks that stem from these lower-level components. To address this problem, McGraw has explicitly introduced the step of dependency analysis (aka "weakness analysis"). One subtask of framework analysis is to check whether security controls (e.g., Security APIs) are correctly applied to implement an application's security requirements. We later discuss our approach based on Design-by-Contract principles, which addresses the problem of employing Security APIs correctly [Mustafa and Sohr, 2015; Sohr et al., 2016].

Having identified possible risks by means of the aforementioned steps, McGraw proposes to prioritize the risks with the help of risk management framework (RMF).

Chapter 3
Architectural Risk Analysis Based on Reverse Engineering Techniques

In this chapter, we introduce approaches to partially reconstructing the security architecture of software systems by means of program analysis techniques. We represent the extracted security architecture in two different forms, DFDs as used in Microsoft's Threat Modeling approach and pre- and postconditions for Java methods [Leavens et al., 2006].

We decided to employ static analyses in our approaches rather than solely relying on dynamic analyses, which are usually used within the process of software penetration testing. Dynamic testing for software security often shows limitations, although software vendors seem to greatly appreciate its value. For example, in one case we reported several *different* TLS vulnerabilities within one single Android application to the vendor's product CERT (security experts). However, they themselves tested the app dynamically by monitoring its connections with the help of the Wireshark tool[12]. Then the product CERT responded that they could not reproduce any reported problem, although three different code locations showed wrong TLS implementations. The problem with their dynamic approach was that it could only show that the connections are encrypted (which was actually the case), but could not reveal that the authentication of the communication endpoints was simply turned off. As a consequence, the app used strong encryption, but it did not know which server it was communicating with. A similar case was reported by Green and Smith [Green and Smith, 2016], hence the need for complimentary tools that help developers and analysts find problems directly in code.

3.1 Reconstructing the Security Architecture in Form of Dataflow Diagrams

In Section 2.3.1 we have indicated that architectural risk analysis is too rarely used in practice because software vendors tend to focus on penetration testing and code analyzers (bug finders)

[12] https://www.wireshark.org/

as the bar is lower to start working on software security. Architectural risk analysis, however, assumes deep security knowledge, which only a few security experts possess [McGraw, 2013]. In particular, it is a mostly manual and therefore time- and cost-intensive process. Bug finders cannot detect such architectural problems as they only concentrate on single program lines and simply find common low-level security bugs, such as buffer overflows, cross-site scripting vulnerabilities and SQL injection problems. Consequently, approaches are desirable that automate the process of architectural risk analysis as well and hence simplify this step for enterprises and organizations. In the following, we present a technique that aims to make architectural risk analysis more tractable in practice.

The main motivation behind our work is the observation that architectural descriptions are often incomplete and do not reflect the actual status of the implementation; in many cases they are even not available at all [Sohr and Berger, 2010]. The idea to automatically reconstruct and analyze the implemented security architecture of software will be even more important when new software trends, such as DevOps (the development of software and its operation grow together), become widely accepted. The main Achilles' heel of DevOps security is the fact that manual architectural risk analysis cannot be adequately carried out in this scenario [McGraw, 2017]. Approaches that automate this process are then of great value to continuously track the state of the current security architecture.

The core idea of our approach was firstly formulated in a position paper [Sohr and Berger, 2010] and thereafter concretized in two BMBF-funded R&D projects, ZertApps (Certified Security of Mobile Application) [VDIVDE-IT, 2014; Bartsch et al., 2014] and ASKS (Architecture-Centric Security Analysis of Business Applications) [DLR, 2010; Berger et al., 2013]. Our technique automatically reconstructs the security architecture from program code. Then this extracted architecture can be visualized, which helps a security reviewer more effectively conduct security analyses. The extracted architecture can also be automatically analyzed by checking security rule sets against it. Our proposed technique differs from current methods of architectural risk analysis, which only work independently of the program code at the level of the software architecture. The practical relevance of this idea is also pointed out by Shostack, the program manager for Microsoft's Trustworthy Computing initiative, who formulates: *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. (…) If technology to do this is further developed, it will present great value to threat modeling.* [Shostack, 2014]

3.1.1 The Core Concepts of our Approach

In the following, we present the core concepts of our approach. This presentation is a high-level description of our technique to give the reader an overview of the main concepts. More details can be found elsewhere ([Berger et al., 2013; Berger et al., 2014; Berger et al., 2016]).

As mentioned before, our main goal is to automatically reconstruct (parts of the) implemented security architecture from program code. Furthermore, we visualize the extracted security architecture such that security experts can obtain a quick overview of security aspects of the analyzed software that are not obvious from merely looking at program code. This is achieved by partitioning the application into components, representing interfaces to external systems explicitly and annotating already implemented security measures. We use DFDs for the visualization of the security architecture as they are used within Microsoft's Threat Modeling process and hence are widely-used in industry [Shostack, 2014].

We employ precise static program analyses, which extract abstract models in form of the DFDs from source code. Our static analyses are implemented with the help of the Soot tool, a framework for static program analysis for Java applications [Vallée-Rai et al., 1999]. Based on Soot and the data structures made available by Soot we have conceptualized and implemented advanced data and control flow analyses to reconstruct a Java application's security architecture. The program-analysis techniques used in this approach include flow-sensitive inter-procedural constant propagation [Berger et al., 2013], the construction of object process graphs (OPGs) [Quante and Koschke, 2008], and inter-procedural program slicing [Krinke, 2003].

The extracted DFDs are then checked against security rules, which are stored in a knowledge base. The rules encoded in the knowledge base describe security flaws at the architectural level. Such flaws, for example, are listed in the CWE [MITRE, 2014] and the CAPEC [MITRE, 2015].

The following example shows a rule, which allows one to check whether confidential data are sent over an unencrypted channel and which corresponds to CWE rule 319 (*CWE-319: Cleartext Transmission of Sensitive Information*) [Berger et al., 2016]:

```
MATCH (src : Element) -[flow : Channel]-> (tgt : Element)
  WHERE flow.type.subtypeof("InterProcessCommunication")
        AND ANY (d IN flow.data WHERE d.IsConfidential)
        AND NOT flow.IsEncrypted
```

On checking such a rule we can identify architectural weaknesses in the design of an application, e.g. an unencrypted channel in a Web application. Technically, this kind of analysis is realized by means of an engine that evaluates the rules on DFDs. More precisely, we translate the DFDs into graphs, which in turn are checked against the rules by the engine [Berger et al., 2016].

3.1.2 Application of the Proposed Tool to Architectural Risk Analysis and Corresponding Use Cases

One main component of our tool is the aforementioned central knowledge base, which contains information about architectural weaknesses in form of rules. This knowledge base can be filled by security and technical experts as well. The encoded knowledge finally simplifies the work of the security analyst.

After the tool has automatically checked the extracted security architecture against the rules in the knowledge base, security analysts or software architects can assess the analysis results. Based upon a prioritization and detailed explanation of the identified problems, next steps to secure the analyzed application can be planned. If possible, the knowledge base can contain information about possible countermeasures. Finally, a risk management framework (RMF) should be applied to conclusively assess the relevance of the discovered architectural weaknesses as proposed by McGraw [McGraw, 2006].

To avoid inaccuracies of the static analyses or to extend the extracted security architecture with information about the IT infrastructure (e.g., network segments, firewalls, and OS versions), for which we do not provide any static program analyses yet, the security architecture can be manually improved by the user. In principle it is even conceivable that the security architecture can be created completely manually and thereafter checked against the rules in the knowledge base. In this specific usage scenario, our tool can be employed within the design phase of the software development process, i.e., no implementation needs to exist.

Since our approach can combine architectural risk analysis with the program code, our tool can be employed from the beginning of coding. The implemented security architecture can therefore be analyzed during the complete implementation phase (see Figure 2).

The main use case, however, is to reconstruct the security architecture from already existing code, where often *no adequate* architectural information is available. Hence, our approach should be foremost applied during the verification phase (see Figure 2).

3.1.3 Current Status of the Tool for Architectural Risk Analysis

Within the frameworks of the BMBF projects ASKS and ZertApps, we implemented the aforementioned concepts as a prototype. In particular, we have built an analysis infrastructure (consisting of more than 400,000 lines of code); this infrastructure allows us to distribute the analyses among different analysis servers such that we can analyze a large set of applications. For the analysis infrastructure, we developed early support of core software frameworks for Java Enterprise Edition systems (JEE) [Berger et al., 2013].

Furthermore, we implemented static analyses for Android applications within the ZertApps project. In the current status, we can extract the single (software) components an Android application consists of (Android activities, services, broadcast receivers, and content providers), and then dataflows between the components can be identified. In Figure 5, we show how the smartphone's device ID flows through a simple Android application and finally leaves the device via an SMS message.

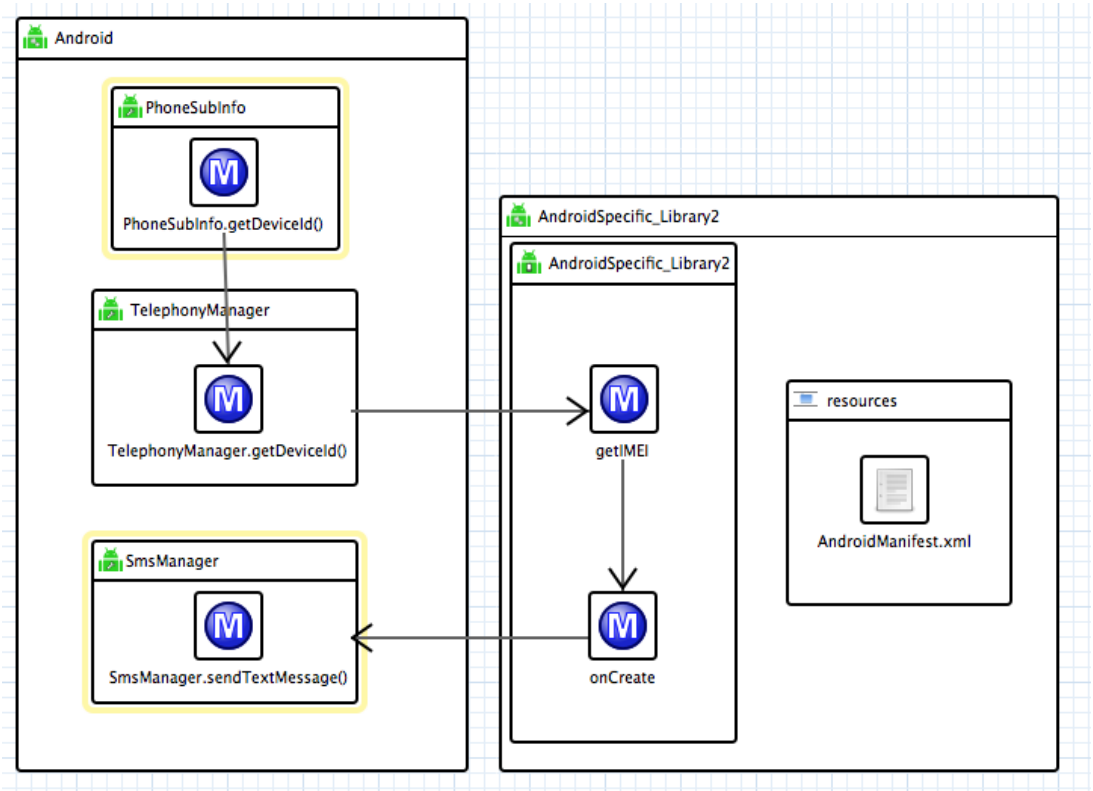

Figure 5: Tracking information flow through an Android application—the device ID is sent (leaked) via SMS.

This existing functionality has also been extended to determine the communication with external servers and to extract information about cryptographic implementations in Android applications [Ziegler, 2015]. Furthermore, advanced Android programming concepts, such as remote service binding, are now supported.

We particularly support hybrid Android applications, which contain both native Android and HTML5/JavaScript code. Figure 6 depicts an example DFD of a hybrid Android application, which uses HTML5. This DFD has been automatically generated with the help of static program analyses from the Android bytecode by employing the Soot framework.

Beyond the static analyses that are used for the reconstruction of the security architecture we have implemented the knowledge base with the security rules. This knowledge base, for example, includes rules to detect unencrypted communication channels for sensitive data, unprotected resources and unprotected external interfaces. More details on the supported rules can be found elsewhere [Berger et al., 2016].

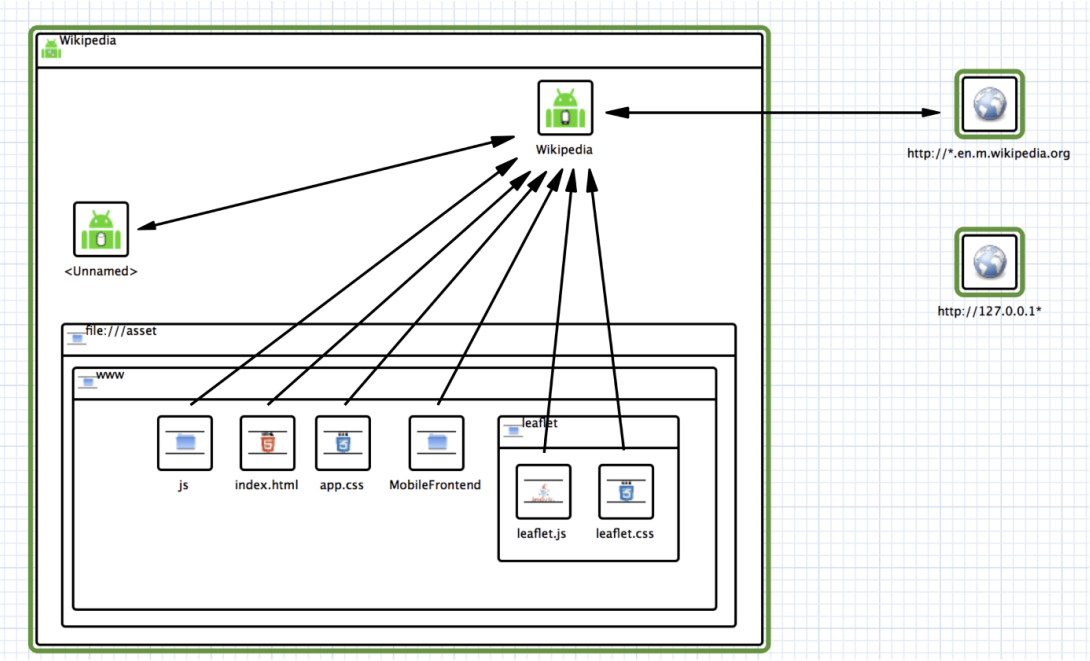

Figure 6: An example DFD extracted from an Android application (figure taken from [Berger et al., 2014]).

3.2 Reconstructing the Security Architecture in Form of Pre- and Postconditions[13]

Until now we have presented a visual approach to architectural risk analysis. In the following, we introduce a complimentary approach, which is based upon formal specifications. After first motivating this specification-based work we discuss its technical details.

As developers tend to use libraries, APIs, and software frameworks to implement their applications, they often employ the security features provided by these frameworks. Security features are then implemented and made available by security APIs (e.g., cryptographic APIs, authentication and authorization APIs). Some APIs, however, do not directly offer security features, but have an impact on the overall security of an application, e.g., network communications APIs. We also regard these security-relevant APIs in the following discussion.

An important subtask of code reviews is to check whether security or security-relevant APIs are used correctly. For example, a study has shown that many popular Android applications implemented SSL functionality wrongly, allowing middleperson attacks [Fahl et al., 2012; Georgiev et al., 2012]. The study concluded that SSL libraries were too complex and hence wrongly used. This problem applies to cryptographic APIs as well, e.g., developers use hard-coded secrets, generate keys with weak entropy or do not know the security implications of selecting a specific cryptographic algorithm or mode [Egele et al., 2013].

As a vision it is desirable to make tools available that simplify security code audits by automatically extracting security-relevant program locations and present them to security reviewers. These tools, in particular, should include a facility to automatically trace important parameters of security/security-relevant API calls to their origin. A reviewer then would not have to tediously search for and extract such code locations. In addition, conformance checking tools, which analyze the extracted code, are very valuable (as far as this is practically possible).

To address this problem, we propose a methodology that replicates security code audits based on known software engineering techniques including program slicing [Krinke, 2003], Design by Contract (DBC) [Meyer, 1989; Leavens et al., 2006], extended static checking [Flanagan et

[13] The following description is based on a technical report published by the TZI [Sohr et al., 2016] and a journal publication [Mustafa and Sohr, 2015].

al., 2002], and annotation inference [Ernst et al., 2007]. Our analysis process starts from security-critical API calls and automatically determines dependences by means of backward slicing. This step, for example, allows security auditors to identify the origin of parameters used in security API calls, e.g., determining the sources of concrete keys and the data to be encrypted. Slicing also makes the step of extended static checking more tractable because code that is irrelevant for security is eliminated and hence the analysis space is smaller.

The use of DBC is twofold. First, it allows us to formulate concise preconditions for security or security-relevant APIs. An extended static checker can then verify whether these preconditions are satisfied by each API call within an application, addressing the aforementioned problem of correct API usage. Second, DBC lets a security architect specify application-specific security requirements in form of postconditions, e.g., an access control policy. Again, an analyst can check whether these security requirements are satisfied by the software with the help of an extended static checker.

Since annotating code with specifications has been shown to be tedious [Flanagan et al., 2002], a method that automatically inserts annotations into program code is useful. For this purpose, we propose to use the Daikon tool, which can infer likely DBC specifications by code instrumentation [Ernst et al., 2007]. The preceding slicing step helps the inference tool produce annotations that better focus on the implemented security mechanisms.

In our approach, we decided to adopt state-of-the-art analysis tools rather than building tools from scratch because we aim to utilize mature base analyses. Other approaches including ComDroid [Chin et al., 2011], CryptoLint [Egele et al., 2013] or MalloDroid [Fahl et al., 2012] in case of Android are specific research tools and do not aim at providing a unified and encompassing solution to the problem of security code audits. Our approach can be applied to different security or security-relevant APIs as we address the general problem of tracing back dependences of security API calls. The slicing part of our technique is independent of the security API under consideration and hence can be easily configured with other security/security-relevant APIs depending on the analysis task. The annotation part, however, must be adjusted to the specific API.

None of the proposed basic analysis tools, however, could be used out of the box. Rather, they had to be appropriately adjusted to fit our purpose. For example, as precise context-sensitive interprocedural slicing is computationally expensive, we had to use optimized, but more imprecise slicing options (e.g., neglecting heap dependences). Then a series of related other slices

had to be added automatically to compensate for the missing precision—identifying the necessary supplemental slices for the specific analysis problem was one of the challenges to be solved. Also, we had to extend the Daikon tool to support exceptional specifications. This extension was finally integrated into the Daikon tool itself as this feature was long missing.

3.2.1 Background

Subsequently, we describe the background technologies, which are the foundation of our work.

Design by Contract

The principle of DBC allows a developer to specify pre- and postconditions, which must be satisfied on function entry and exit, respectively [Meyer, 1989]. Invariants apply to the entry and exit of all *public* methods. For most mainstream programming languages, DBC extensions exist [Burdy et al., 2005; Chess, 2002; Leino, 2009], most notably, the Java Modeling Language (JML) [Leavens et al., 2006]. Furthermore, some specification languages, which are independent of a particular programming language, also support DBC concepts, such as the Object Constraint Language (OCL) [Warmer and Kleppe, 2003]. A comprehensive overview of DBC-based specification languages can be found in a survey by Hatcliff et al. [Hatcliff et al., 2012].

We describe JML subsequently in more detail because we use it for the discussion of our technique throughout the rest of this chapter.

Java Modeling Language

JML is a formal behavioral interface specification language, specifically designed for specifying the functional behavior of Java programs [Leavens et al., 2006]. Due to the fact that JML specifications are written by the Java programmers themselves at the source code level, JML uses a Java-like syntax and is relatively easy to understand by an average programmer.

JML provides a rich set of language constructs that are necessary to precisely specify the functional behavior of Java programs, mostly, in the form of class invariants as well as pre- and postconditions of methods. JML specifications are written in special annotation comments in the form of `/*@...@*/` or simply `//@...` if a single line specification is intended. The JML tools use these annotations to parse the JML specifications out of the Java programs. JML provides `requires` and `ensures` clauses to specify pre- and postconditions of a method.

The preconditions enforce the client's obligations, whereas postconditions enforce the implementer's obligations. JML provides a logical variable `\result`, which represents the value returned by a method. In addition, JML supports the concept of **pure methods**, which are side-effect free public methods. Only these kinds of methods can be called within a JML specification.

Extended Static Checking

A variety of tools exist that allow one to check the JML constraints at run-time or statically [Burdy et al., 2005]. One such tool is ESC/Java2, which statically detects inconsistencies between the code and the specification using a built-in automatic theorem prover. However, since such conformance checking is in general undecidable, false positives and negatives may be produced. ESC/Java2 employs **modular reasoning**, which is an effective technique when used in combination with static checking. Java methods are analyzed one at a time and their JML-based specifications can be proven by inspecting only the specification contracts (and not the code) of the methods called within their bodies [Flanagan et al., 2002].

To improve extended static checking, there are currently ongoing efforts for building a new extended static checker for Java within the OpenJML initiative [Cok, 2011]. At the time of this writing, however, this tool does not support reasoning on Strings, as David Cok, the architect of OpenJML, concedes:

OpenJML (and solvers) don't support reasoning about Strings at present. New versions of SMT-LIB have a built-in theory for strings, so I'm hoping to use that when solvers support it. [Cok, 2017].

We repeatedly use Strings within our JML specifications as, for example, can be seen in Figure 10. Current SMT solvers, which form the basis of OpenJML's ESC tool, start to support such String theories, for example, CVC4 [Liang et al., 2016]. Therefore, we have hope that our approach can be applied to larger case studies in industrial contexts in the near future.

Program Slicing

Program slicing was first introduced by Weiser who pointed out that developers understand programs according to dependences between statements and not necessarily according to the natural order of the code [Weiser, 1981]. A backward slicing algorithm starts from a statement, the so-called "slicing criterion", and calculates all the statements that (transitively) influence

it. Slicing is often used for program comprehension and debugging tasks in order to focus on those code parts that are relevant for the analysis. Technically, slicing is usually implemented by system dependence graphs (SDGs) [Horwitz et al., 1990]. SDGs often contain the statements in static single assignment form (SSA) [Appel and Palsberg, 2003], an intermediate representation well-suited to data and control flow analyses, as well as call graph information. In particular, an SDG represents methods via special nodes. Context-sensitive slicing only allows accessible execution paths, i.e., a method must return to the site where the method has been called and not to other call sites of the method. Krinke gives a detailed overview of slicing techniques [Krinke, 2003].

3.2.2 The Core Concepts of Our Approach

The basic idea of this approach is as follows. First, we build an SDG of the software under analysis, which, for example, is a standard task for static analysis frameworks, such as Soot and WALA [Dolby and Sridharan, 2010]. On this graph-based data structure we search for security/security-critical API calls. A typical security-API call is `EJBContext.isCallerInRole()` provided by the JEE framework, which checks whether the calling user of a Java-based Web application has assumed a specific role (authorization check). Other examples are the `Cipher.doFinal()` method, which encrypts or decrypts data, and `Signature.sign()` for the generation of digital signatures. Security-relevant APIs include communications APIs, e.g., `URLConnection.connect()`. Per se, the `connect()` method cannot be considered a security API. However, it is relevant to security because it opens a connection, which in many cases must be secured by additional mechanisms, such as TLS encryption [Dierks and Rescorla, 2008]. Table 1 shows typical security(-relevant) API calls, which can be used as starting points for different analysis tasks.

We reconstruct the implemented security architecture (or more precisely, parts of it) by performing a backward slicing step, with security/security-relevant API calls as slicing criteria (seed statements). This step allows us to automatically determine the origin of the used data, e.g., encryption keys, data to be encrypted or communication channels to be protected by TLS. Having automatically extracted this information from program code, an analyst can use these slices for security code audits and better comprehend the implemented security feature as slicing reduces the code size. It has been shown experimentally that slices only contain 30% of the original code on average [Krinke, 2003]. The reduced code size also makes later steps of our

approach, such as annotation inference and extended static checking, feasible. In order to follow the slice backwards through method calls, we employ inter-procedural slicing [Horwitz et al., 1990; Krinke, 2003].

API	API class	Seed method calls
JEE authorization API	EJBContext	isCallerInRole()
Servlet authentication	HttpServletRequest	login(), logout()
Java encryption	Cipher	doFinal()
Java signature	Signature	sign(), verify()
Java crypt hash	MessageDigest	digest()
Java keystore	KeyStore	setKey(), set*Entry()
Https-based communication	HttpsURLConnection, HttpURLConnection	connect()

Table 1: API calls to be used as slicing seeds.

In a second step, the sliced code is annotated with JML specifications, in form of pre- and postconditions. This can either be done manually—which may sometimes be an error-prone and tedious process in more complex software—or automatically by an annotation detector, such as Daikon [Ernst et al., 2007]. Daikon infers likely annotations by using code instrumentation techniques.

We presume that the security APIs themselves have already been annotated with JML specifications. This preparing step in essence is mostly manual, but needs to be done only if a new library is to be annotated or an existing and already annotated library has been modified. One example of such an annotated library class is depicted in Figure 11. In this case, the `javax.crypto.Cipher` class has been annotated with JML specifications.

We propose to utilize the advantages of DBC-based specifications because we consider the application of security APIs a contract between the client (the application) and the callee (the security library/security API). The precondition of a security API method must be satisfied by the client, whereas the postcondition must be fulfilled by the called API method. Preconditions

help a developer use security APIs correctly, e.g., employing state-of-the-art cryptographic algorithms or generating secure random numbers for a symmetric key. The postconditions must then be strong enough to enforce an application's security requirements, e.g., the intended access control policy. Consequently, the client is responsible for selecting appropriate security APIs and using them in a way to guarantee the application's security requirements.

After annotating the software under analysis, the code can be checked against the annotations. One possibility is extended static checking with tools like ESC/Java2 [Burdy et al., 2005]. Proceeding this way, discrepancies between code and specifications can be detected. Figure 7 recapitulates the single steps of our approach.

High-level algorithm: Static Analysis of an Application against its Security Requirements

Input: The sources of an application and the employed security APIs.

Step 0: Annotate the Security APIs with DBC specifications.

Step 1: Load the application's source code into an SDG to enable program analysis.

Step 2: Search for calls of the Security API on the SDG and add them to the slicing criterion (automated step).

Step 3: Do context-sensitive, inter-procedural backward slicing w.r.t. the slicing criterion.

Step 4: Create a new source file with the sliced version of the application.

Step 5: Insert DBC annotations for each method, possibly by employing an automated inference mechanism, such as Daikon.

Step 6: Call an extended static checker on the annotated code.

Result: A static checker's report on specification violations of the specified security requirements.

Figure 7: Overall algorithm for abstracting and analyzing the implemented security architecture of an application (figure taken from [Sohr et al., 2016]).

In summary, our approach replicates and automates the procedure security analysts follow when auditing code. Through the automated extraction of security-relevant parts from the code, the process of security reviews can be better supported. Security evaluators usually start their

analysis from security-relevant API calls, such as `Cipher.doFinal()` or `UrlConnection.connect()`. Then they trace back parameters, return values and further related objects to their origins while carrying out manual security analyses (quasi in their mind based on experience). The step of tracing back security-relevant calls and their parameters is simulated by the backward slicing step. The security knowledge of security experts is encoded in a knowledge base of JML annotations and thereafter security analyses are carried out by the extended static checker. In principle, this knowledge base in conjunction with extended static checking complements and partly replaces the job of security experts. As the knowledge base covers different security aspects, common situations can be avoided where one security aspect has been considered in great depth, whereas others have been neglected. For example, SAP's Android applications showed quite good cryptographic implementation (reasonable key management, usage of secure algorithms, secure random number generation), but weaknesses in SSL/TLS encryption. In this case, expert knowledge concerning SSL/TLS libraries was apparently missing.

The DBC-based approach can be applied to handle situations like the service spoofing vulnerability, which has been discussed in Section 2.3.1. For example, we can define a precondition for the Android `Context.startService()` method (and also for `Context.startActivity()`) to ensure that a service is only called if it has been signed by the designated developer (and not by an attacker):

```
/*@
  requires service.getPackage() != null  ==>
             this.getPackageManager().checkSignatures(
                this.getPackageName(),service.getPackage())
                                == PackageManager.SIGNATURE_MATCH;
@*/

ComponentName startService(Intent service){
     ...
}
```

The called service cannot be spoofed by an attacking service with the same name because it must be signed with the same certificate as the calling app—an attacker does not possess the corresponding private key, which is needed to perform the attack. We do not discuss this point in more detail here, but go on with a discussion of two case studies from the JEE context as well as Android OS.

3.2.3 Application to JEE-Based Software

We now discuss our technique with the help of a fictitious clinical information system, which is to be evaluated by internal quality assurance (QA) or an external evaluator (e.g., with respect to the Common Criteria [Common Criteria, 2009]). Among other functionality, it should provide means for reading and writing electronic health records (EHRs). We further assume that it is a JEE-based client-server application.

Figure 8 depicts some security requirements for this application as part of a hospital security policy. Req 3 requires that a clinician must play the role *Physician* to read or write the patient data of the EHR; for reading prescriptions, the role *Nurse* suffices (Req 1). There are additional access control requirements, which state that the clinician must be on the same ward as the patient—these are usually called "context constraints" [Georgiadis et al., 2001]. The hospital security policy also comprises confidentiality and integrity requirements, such as "a doctor's letter must be encrypted and digitally signed with the treating physician's certificate" (Req 5). A doctor's letter is usually sent to a general practitioner after a patient's treatment at a hospital has been finished and a follow-up treatment has been necessary. As the hospital belongs to a healthcare provider, who runs several hospitals, storage capacity as a cloud service is offered to the hospitals. The corresponding connections must be appropriately authenticated and secured (Req 7).

Excerpt from a security policy of a clinical information system

Req 1: Data about a patient's prescriptions may only be read by clinicians who assume the roles *Physician* or *Nurse* and are on the same ward as the patient.

Req 2: Data about a patient's prescriptions may only be written by physicians who are on the same ward as the patient.

Req 3: Patient data may only be read or written by clinicians with the role *Physician*.

Req 4: Patient data may only be written or read by physicians who are on the same ward as the patient.

Req 5: A doctor's letter must be encrypted and digitally signed with the treating private key.

Req 6: Sending a doctor's letter is only allowed for clinicians with the role *Physician*.

Req 7: The communication with a Cloud-based server must be authenticated, and confidentiality as well as integrity of the sent/received data must be assured.

Figure 8: Security requirements of the clinical information system (see also [Sohr et al., 2016]).

To illustrate our ideas, Figure 9 depicts an excerpt of the implementation of a fictitious clinical information system (a more complete version is given in our technical report [Sohr et al., 2016]). The implementation employs JEE's programmatic authorization `EJBContext.isCallerInRole()` to enforce the requirement that the caller of a method plays the appropriate roles before calling security-critical code. For example, the method `writePrescriptions()` checks whether the caller has already assumed the role *Physician*. In addition, the code uses cryptographic functionality, which is provided by Java security libraries. Also, TLS/SSL functionality is implemented using Java security and Apache libraries to enable secure communications with the Cloud. We assume that the developers implemented TLS/SSL functionality because they provided their own X.509 root certificate stored in a Java keystore as a certificate-pinning approach (see Figure 9).

The job of software QA is to evaluate the software with respect to the security requirements presented in Figure 8. Typical questions to be answered by an analyst include "Does the `writePrescriptions()` method satisfy Req 2?" and "Does the `sendDiagnosis()` method make sure that patient data are encrypted (Req 5), and if so, are secure encryption algorithms used?". It is often a laborious task for an evaluator to understand the details of the code and to identify the relevant code parts which implement an application's security architecture. In practice, the code is much more complex than our example. Here our approach comes into play.

Subsequently, we discuss the single steps of our approach with the help of several code examples.

3.2.3.1 Discussion of the Approach with the Help of Code Examples

Figure 10 shows the sliced and annotated code from Figure 9. We used the three slicing criteria `ctx.isCallerInRole("Physician")`, `mCipher.doFinal(data)`, and `urlConnection.connect()` reflecting the three different security aspects authorization, data encryption, and communications security. Please note that after slicing, some code including several auxiliary methods (e.g., `getMailAddress()`, `getExternalPractice()`, `isConfirmedSendDiagnosis()`) is not contained in the slice anymore. This code is not interesting for our specific analysis purposes.

After slicing the code, JML annotations are inserted (either manually or automatically by a tool such as Daikon). For example, the authorization requirement "Data about a patient's prescriptions may only be written by physicians who are on the same ward as the patient." (Req 2) is shown in Figure 10 as a JML specification. The method returns normally if the *Physician* role has been activated and the additional context constraint ("the same ward") is satisfied; otherwise, a security exception is thrown. Similarly, one can provide annotations regarding encryption and communications security.

Thereafter, an extended static checker, such as ESC/Java2 or OpenJML's ESC, can be used to verify whether the code actually satisfies its JML specifications. In particular, the extended static checker can utilize specifications of the employed security APIs for the verification task. For example, the postcondition of the `encryptData()` method (see Figure 10) can be proven by using the annotations of the `Cipher` class depicted in Figure 11—these example specifications may have been stored in a knowledge base by a security expert before. In the given example, checking the preconditions implies that `Cipher.init()` and `Cipher.getInstance()` must have been called before with the appropriate parameters for the key and the encryption algorithm.

Please notice that the annotations for the Java `Cipher` class use the concept of **implementation stubs**. The background is here that we do not intend to validate the implementation of Java libraries. Rather, our goal is to assure that the libraries are used correctly or in a way that an application's security requirements are satisfied. In particular, we utilize the assume statement [Flanagan et al., 2002], which is built into ESC/Java2 and unconditionally assumes that a specification is fulfilled without checking the code (which deliberately is only a stub in our case).

The example code in Figure 9 contains several vulnerabilities. The method `sendDiagnosis()` does not implement Req 6, i.e., no access control checks exist and hence access is allowed for all participants of the clinical information system regardless of their roles.

Furthermore, the hostname verifier in the TLS code is set to an instance of `AllowAllHostnameVerifier`. This class essentially turns TLS hostname verification off. Even software vendors with a well-defined SDL follow such practices, e.g., SAP, who built this code into a mobile communication library such that several (security-critical) apps were vulnerable. To detect such flawed TLS code, the `setHostnameVerifier()` method of the `HttpsURLConnection` defines a precondition as follows:

```
/*@ requires
      v instanceof StrictHostnameVerifier ||
      v instanceof BrowserCompatHostnameVerifier;
 @*/
```

This precondition assures that the full implementation for hostname verification is used.

A further vulnerability can be found in the following statement (see Figure 9)

```
mCipher = Cipher.getInstance("AES");
```

In this code, neither the encryption mode nor a padding scheme is defined. Depending on the pre-installed Java crypto provider, the electronic code book mode (ECB) could be the default, which is known to be insecure [Anderson, 2008].

If the method `Cipher.getInstance()` contains the precondition

```
//@ requires \result.getAlgorithm().equals("AES/CBC/PKCS5Padding");
```

then an extended static checker can automatically identify the aforementioned issue.[14]

As a further observation, (a library's) preconditions tend to correlate to rules that guarantee the secure usage of an API method. For example, the `getInstance()` method has a precondition stating that the more secure CBC (cipher block chaining mode) [Anderson, 2008] should be used with AES encryption. Similar remarks apply to the precondition for the `setHostnameVerifier()` API method. Since preconditions are required to be satisfied by the caller, they can be conveniently provided by the called library (e.g., in a knowledge base as discussed above).

Postconditions, such as exceptional specifications and `ensure` statements, let one express application-specific security requirements. Application-specific requirements include the role-based policy or requirements stating which data are to be encrypted or signed by which key.

[14] Since the CBC mode only provides confidentiality rather than authenticity, security experts now recommend the Galois/Mode (GCM) [McGrew and Viega, 2004]. GCM provides both security objectives and is supported by many Java crypto providers. The JML specification can then be adjusted accordingly with the constraint `\result.getAlgorithm().equals("AES/GCM/NoPadding")`.

```
@Resource EJBContext ctx;
Cipher mCipher;

// public interface

public void writePrescriptions(String ehrID, String userID, String pres){
      Clinician clinician = getClinician(userID);
      EHR eHR= getEHR(ehrID);
      if !(ctx.isCallerInRole("Physician")
                  && eHR.getWard() == clinician.getWard())
           throw new SecurityException("No sufficient access rights.");
       eHR.setPrescriptions(pres);
}

public void sendDiagnosis(String userID, String ehrID, Key key){
      EHR eHR = getEHR(ehrID);
      Clinician clinician = getClinician(userID);
      byte[] encryptedDiagnosis=
                        encryptData(eHR.getDiagnosis().getBytes(), key);
      String externalPractice = getExternalPractice();
      String mailAddress = getMailAddress(externalPractice);
      if(isConfirmedSendDiagnosis())
         sendPatientData(encryptedSignedDiagnosis, mailAddress);
      Logger.log();
}

public void conncectToHealthcareProviderCloud(URL url) throws IOException,
GeneralSecurityException {
      TrustManagerFactory tmf = TrustManagerFactory.getInstance("X509");
      tmf.init(getKeystore());
      SSLContext context = SSLContext.getInstance("TLS");
      context.init(null, tmf.getTrustManagers(), null);
      HttpsURLConnection urlConnection =
            (HttpsURLConnection) url.openConnection();
      urlConnection.setSSLSocketFactory(context.getSocketFactory());
      urlConnection.setHostnameVerifier(new AllowAllHostnameVerifier());
      urlConnection.connect();
}

// helper methods

byte[] encryptData(byte[] data, Key key){
      mCipher = Cipher.getInstance("AES");
      mCipher.init(Cipher.ENCRYPT_MODE, key);
      return mCipher.doFinal(data);
}

String getMailAddress(){…}

String getExternalPractice(){…}

boolean isConfirmedSendDiagnosis(){…}

void sendPatientData(byte[] encryptedDiagnosis, String mailAddress){…}
```

Figure 9: Excerpt from the source code of a fictitious clinical information system (see also [Sohr et al., 2016]). The slicing criteria are underlined.

```
/*@ public normal_behavior
      requires ctx.isCallerInRole("Physician") &&
        getEHR(ehrID).getWard() == getClinician(userID).getWard();
    also
    public exceptional_behavior
      requires !(ctx.isCallerInRole("Physician")  &&
        getEHR(ehrID).getWard() == getClinician(userID).getWard());
      signals_only SecurityException;
 @*/
public void writePrescriptions(String ehrID, String userID, String pres){
      Clinician clinician = getClinician(userID);
      EHR eHR= getEHR(ehrID);
      if !(ctx.isCallerInRole("Physician") &&
                                  eHR.getWard()== clinician.getWard())
            throw new SecurityException("No sufficient access rights.");
}


/*@ public normal_behavior
      requires ctx.isCallerInRole("Physician")  &&
        getEHR(ehrID).getWard() == getClinician(userID).getWard();
      ensures
         mCipher.getInput().equals(getEHR(ehrID).getDiagnosis().getBytes())&&
         mCipher.getKey().equals(key) &&
         mCipher.getAlgorithm().equals("AES/CBS/PKCS5Padding");
   also
   public exceptional_behavior
      requires !(ctx.isCallerInRole("Physician")  &&
        getEHR(ehrID).getWard() == getClinician(userID).getWard());
      signals_only SecurityException;
 @*/
public void sendDiagnosis(String userID, String ehrID, Key key){
      EHR eHR = getEHR(ehrID);
      Clinician clinician = getClinician(userID);
      byte[] encryptedDiagnosis =
                        encryptData(eHR.getDiagnosis().getBytes(), key);

}


/*@ ensures mCipher.getInput().equals(data) && mCipher.getKey().equals(key)
            && mCipher.getAlgorithm().equals("AES/CBS/PKCS5Padding")
            && mCipher.getOutput().equals(\result);
@*/
byte[] encryptData(byte[] data, Key key){
      mCipher = Cipher.getInstance("AES");
      mCipher.init(Cipher.ENCRYPT_MODE, key);
      return mCipher.doFinal(data);
}
```

Figure 10: Annotated and sliced code from Figure 9 (only excerpt).

```
public class Cipher{

 /*@
     requires algorithm.equals("AES/CBS/PKCS5Padding");
     ensures  \result.getAlgorithm().equals(algorithm) &&
                          algorithm.equals("AES/CBS/PKCS5Padding");
   @*/
 public static Cipher getInstance(String algorithm}{
   Cipher res = new Cipher();
   //@ assume res.getAlgorithm().equals(algorithm);
   return res;
 }

 //@ ensures this.getKey().equals(key);
 public void init(int mode, Key key){
   //@ assume this.getKey().equals(key);
 }

 /*@
    ensures this.getInput().equals(data) && this.getOutput().equals(\result);
   @*/
 public byte doFinal(byte[] data){
   byte[] res = new byte[42];
   //@ assume this.getInput().equals(data) && this.getOutput().equals(res);
   return res;
```

Figure 11: Implementation stubs for the `javax.crypto.Cipher` class—JML annotations for security libraries can be stored within a security knowledge base.

3.2.3.2 Early Experience of our Approach within an Internal Project

In the context of a bachelor thesis, we have implemented a JEE application to obtain early feedback if and how our techniques work, specifically with a focus on the slicing part [Gerken, 2015]. We also used this example application to construct about 30 small test cases, which allowed us to evaluate the slicing approach, which has been implemented by means of IBM's WALA tool [Dolby and Sridharan, 2010].

Although we demonstrated that our techniques worked in principle, we found out that the slicing approach added nearly all Java API method calls even if they are not related to the analyzed problem at all. This led to false positives in the sense that the slices contained code that was not relevant to security. We could also track back the reason for this behavior: The slicer correctly assumed that Java method calls potentially threw exceptions, which in turn meant that these method call statements controlled subsequent statements (as control dependencies). Hence nearly all Java API calls appeared in a slice. One solution to this problem is to use WALA's `NO_EXCEPTIONAL_EDGES` option. This option, however, sporadically did not

work correctly (code was missing, which led to false negatives) [Detmers, 2016]—this bug is currently being discussed with the WALA developers with the help of a bug report [Sridharan, 2017].

At any rate, false positives are not as problematic as false negatives because the sliced code only contains additional statements. More details on this point are given in our technical report [Sohr et al., 2016].

3.2.4 Application to System Services of the Android Platform

We also applied our approach to the Android platform. We only sketch the results of this work here, whereas the details can be found in a journal paper, which is also part of this thesis [Mustafa and Sohr, 2015].

The background of applying our technique in this context was to analyze Android system services. The Android operating system makes available 40 system services, which provide security-critical core functionality of Android. Among these system services are the `PackageManagerService` (administers all Android app packages), the `AccountManagerService` (manages accounts on Android devices, such as Google, Facebook, Skype, and Twitter accounts), the `BluetoothService` (responsible for Bluetooth access), and the `ActivityManagerService` (manages runtime-behavior of Android applications).

Most of these services can be accessed from Android applications with the help of specific APIs. Access to these APIs, however, is restricted by permissions. These permissions must either be granted at installation time by the end user or by the Android system itself based on the Android app's signature (in case of very dangerous permissions).

The Android system services enforce the permissions by specific authorization checks (aka authorization hooks), which are usually enforced before a system service's critical functionality is executed [Enck et al., 2009]. Figure 12 shows an example of a sliced and annotated Android API method (taken from the `DevicePolicyManagerService`, which can be called by device administration apps). In particular, one can see that Android uses the `enforceCallingOrSelfPermission(perm)` API call, which in turn calls a permission check `checkCallingOrSelfPermission(perm)` within its implementation. `checkCallingOrSelfPermission(perm)` checks whether the caller has been granted the permis-

sion `perm`—in this case the `MANAGE_DEVICE_ADMIN` permission. `enforceCallingOrSelfPermission(perm)` then enforces the access decision. About 500 similar authorization checks exist in Android system services, leading to a complex access control system.

The overall goal of our work is the redocumentation of the access control policy implemented in the Android platform. This policy is mostly undocumented, although it is important for an external security expert to comprehend Android's security mechanisms. Only the source code of the services is available, but no further architectural documents. Furthermore, many APIs are hidden, i.e., they can be called by a developer through Java reflection techniques, but they should never be used in an application [Porter Felt et al., 2011]. Our approach allows a security auditor to make explicit the access control policies for system services including the hidden functionality.

We have implemented a tool that automatically extracts the aforementioned authorization hooks for ten selected system services of Android version 4.0.3 [Mustafa and Sohr, 2015]. Later, the tool has been extended within the context of a master thesis to extract these hooks from all Android versions available at that time [Gulmann, 2014].

The implementation of this tool is based on the concepts that have been introduced in Section 3.2.2. We automatically collect all authorization hook calls via searching the SDG with a depth-first search and then apply the WALA slicer. We also implemented a GUI feature that lets an analyst insert JML annotations into the sliced code similar to the annotation depicted in Figure 12. Thereafter, we applied ESC/Java2 on this annotated and sliced code to check for discrepancies between the Java code and the JML annotations. Due to engineering issues, we could only apply the tool chain to selected Android system services. As a result, we found three implementation errors in the access control policy of the `BluetoothService` of Android version 4.0.3 [Mustafa and Sohr, 2015].

The main result, however, was to make available a tool that lets a security analyst automatically redocument the implemented access control policy of Android system services in form of DBC specifications. This redocumented policy can then be reviewed by external auditors, e.g., from authorities such as the German Federal Office for Information Security (BSI), to better understand security mechanisms implemented in Android. This experience can finally be used for a

broader IT risk management of the Android platform. In particular, with the help of our approach, an assessment of the Android platform with respect to its suitability for security-critical application scenarios, e.g., in governmental contexts, is conceivable.

```
/*@
   public normal_behavior
      ensures (this.mHasFeature == false) ==>
       (this.mContext.checkCallingPermission(MANAGE_DEVICE_ADMINS)
                                                == PERMISSION_DENIED
        || this.mContext.checkCallingPermission(MANAGE_DEVICE_ADMINS)
                                                == PERMISSION_GRANTED);
   also
   public exceptional_behavior
      requires
       this.mContext.checkCallingOrSelfPermission(MANAGE_DEVICE_ADMINS)
                                                    == PERMISSION_DENIED;
      signals_only SecurityException;
@*/

public   void   setActiveAdmin(ComponentName   adminReceiver,   boolean
refreshing, int userHandle) {
      if (!mHasFeature) return;
      mContext.enforceCallingOrSelfPermission(MANAGE_DEVICE_ADMINS,null);
}
```

Figure 12: A sliced and annotated Android API method from the Android system service `DevicePolicyManagerService`.

3.3 Summary and Outlook of this Work

In this chapter, we presented two approaches that aim to reconstruct the implemented security architecture from Java-based software. One of the proposed techniques extracts the implemented security architecture in form of DFDs, which allows us to visualize the architecture and thereafter analyze it (partly) automatically. This approach has been developed within four different R&D projects, ASKS, ZertApps, CertifiedApplications (funded by the German Federal Ministry for Economic Affairs and Energy, BMWi) and a research contract by a large enterprise (see Chapter 5). Currently, the developed prototype consists of ca. 400,000 lines of code and supports Android, Java and JEE as frameworks/programming languages. Within the CertifiedApplications project the prototype was enhanced to integrate further Java-based security concepts, such as Java's cryptographic APIs; also other Java Web frameworks, such as JavaServer Faces, are now better supported to allow for more comprehensive security analyses.

In addition, the PortSec project has recently been finished, which aimed to develop a comprehensive IT risk management framework for port community systems (PCS) [VDI/VDE-IT,

2016, Sohr and Maeder, 2019]. This project allowed us to integrate application-specific security requirements into our approach, i.e., we covered security requirements from the domain of port logistics [Berger et al., 2019]. Also, the BMBF-funded project SecureSmartHomeApp was carried out at the TZI dealing with the security analysis of app-controlled smart home systems. Again, our tool for architectural risk analysis was extended and evaluated in a further interesting application domain [Sohr and Schröer, 2019].

The second approach, which attempts to regain DBC specifications from Java software, is not as mature as the aforementioned work. We currently do not have any funding for this approach, which jeopardizes the success somewhat. However, our early results are quite promising [Mustafa and Sohr, 2015] and a recently finished master thesis at the TZI hints at this direction [Cyl, 2019]. In particular, we hope to develop a tool that can analyze the Android platform more comprehensively than the current demonstrator does. In the meantime, other works have been published, which deal with the topic of better understanding the Android platform itself and its implemented access control policy rather than analyzing Android applications [Backes et al., 2016; Shao et al., 2016; Gorski et al., 2019].

In summary, the main contribution of this thesis to both approaches lies in defining the overall research goals, i.e., combing static program analyses with specific aspects from the field of software security. In particular, Threat Modeling [Shostack, 2014] can be more effectively applied by our approach; aspects of understanding cryptographic code and implemented access control policies can now also be handled. Furthermore, it remains to be seen how far the visual and the specification-based approach complement each other in practice.

Within the frameworks of our research at the TZI, we followed another approach that combines static code analyses with software security. In this work, we extracted manifestations of low-level security patterns from Android applications, e.g., cryptographic implementations, authentication enforcement, or secure communication channels with TLS. By identifying security-pattern instances in program code we address the point of security program comprehension; also the results of this security analysis can be used to measure the security level of the analyzed software. More information on this approach can be found in a journal paper from Bunke and Sohr [Bunke and Sohr, 2020], which is also a part of this habilitation thesis.

Chapter 4
Managing and Analyzing Role-Based Policies with UML and OCL

This chapter introduces our forward-engineering approach to architectural risk analysis by defining systematically role-based policies. As RBAC is a policy-neutral access control model [Sandhu, 1996; Sandhu et al., 1996], it can be used for different kinds of applications with quite different access control policies, ranging from banking software and e-government applications to clinical information systems. With RBAC and advanced concepts, organisation-specific access control requirements can be implemented. Before introducing our approach to modeling and analyzing role-based policies, we give a brief overview of RBAC and its extensions.

4.1 Role-Based Access Control and its Advanced Concepts

As studies have shown, role-based access control (RBAC) simplifies access management in organizations [O'Connor and Loomis, 2010]. The basic idea of RBAC is to assign access rights (permissions) to users via roles rather than directly assigning access rights. Roles usually correspond to job functions, which employees may have within their organization. In a hospital, roles such as *physician*, *nurse*, and *administrative* are conceivable, whereas in a financial institute one may define roles like *cashier*, *cashier supervisor*, or *accountant*.

One can also differentiate between role assignment, where an administrator assigns roles to users (e.g., employees in an enterprise), and role activation. Role activation accounts for the principle of least privilege, where a user only activates those roles that are necessary for successfully carrying out her job.

Often the concept of role hierarchies is useful in hierarchically-structured organizations, i.e., senior roles exist, which inherit permissions from junior roles. For example, the role *department head* may inherit from the role *employee*.

4.1.1 RBAC96

Based on these concepts, Sandhu et al. defined the RBAC96 model [Sandhu et al., 1996], which later influenced the RBAC ANSI standard [American National Standards Institute, 2004]. The ANSI RBAC model comprises the following sets and relations:

- the sets *Users*, *Roles*, *Permissions*, *Objects*, *Operations*, and *Sessions*,
- user assignment*: UA: Users x Roles*,
- permission assignment: *PA: Permissions x Roles*,
- role hierarchy: *RH: Roles x Roles.*

Objects are the entities to be accessed by users of the system, e.g., files, database tables, accounts, or electronic health records. Operations, such as read, write, debit, and credit, can be executed on objects. Permissions are usually pairs of operations and objects, e.g., (read, myFile), (write, yourFile), or (debit, myAccount). Finally, the session concept reflects the least privilege principle, i.e., roles must be activated explicitly within sessions.

4.1.2 Authorization Constraints

One of the most important advanced RBAC concepts are role-based authorization constraints. Sometimes authorization constraints are said to be the main motivation behind the introduction of RBAC [Sandhu et al., 1996]. Authorization constraints represent additional access control rules relevant to an organization. In the following, we give several examples of such organizational rules:

- Separation of Duty: Separation of Duty (SoD) requires one to split job functions to prevent fraud and error [Simon and Zurko, 1997; Sandhu et al., 1996; Gligor et al., 1998]. For example, a cashier may not act as a cashier supervisor at the same time; in a hospital setting the roles physician and pharmacist must be separated. One common way to implement SoD is to define **mutually exclusive roles** [Ferraiolo et al., 2007].

  One can differentiate between static SoD (SSD) and dynamic SoD (DSD). Static SoD is enforced when users are assigned to roles, e.g., by an administrator. In contrast, dynamic SoD is less restrictive in that it allows exclusive roles to be assigned to one user;

however, this user is not allowed to activate all these roles while the application is running.

In literature, more fine-grained SoD rules have been discussed. For example, Object-Based Dynamic SoD prevents a user from executing more than one task of a business process against an object, e.g., a user may not prepare a cheque and thereafter also verify it [Simon and Zurko, 1997; Gligor et al., 1998; Nash and Poland, 1990]. History-Based Dynamic SoD is similar to Object-Based Dynamic SoD, but it prevents a user from executing all transactions against an object, e.g., an employee must not fill in her travel application, approve this application, and check whether the financial budget for travelling is available.

- Cardinality Constraints: Cardinality constraints impose numerical limits on RBAC relations, such as UA and PA [Sandhu et al., 1996]. One typical example of a cardinality constraint is to require that at most one user may be assigned to the role *department head*.

- Prerequisite Roles: The prerequisite roles authorization constraint requires a user to also be assigned to a specific further role when she is assigned to the intended role [Sandhu et al., 1996]. For example, if a user is assigned to the *head of department* role, she must also be assigned to the *employee* role.

- Context Constraints: Context constraints are an important additional access control concept in organizations [Georgiadis et al., 2001]. Typical contexts are location (e.g., a clinician may only have access to electronic health records of patients who are on the same ward) or the specific case currently being considered (e.g., a clinician may only access electronic health records if she is at the same time assigned to the corresponding patient).

- Temporal Constraints: Temporal constraints are also a special case of context constraints [Joshi et al., 2005]. A typical temporal constraint is a role which can only be activated at specific periods of time (e.g., the *emergency medical service* role can be assumed at night or on the weekend).

Due to their relevance role-based authorization constraints are an important aspect of the work described in the following subsections. In particular, we support dynamic SoD constraints, such as Object-Based Dynamic SoD [Kuhlmann et al., 2011; Kuhlmann et al., 2013].

4.2 Modeling and Validating RBAC Policies with UML and OCL

It is difficult to define consistent role-based policies in an organisation as they can become a complex rule set [Sohr et al., 2008]. In many organizations such policies are even not made explicit and furthermore it is unclear whether they have been actually implemented in application software.

Therefore, this part of the habilitation thesis deals with the topic of how to specify role-based policies in a formalism to make the policies explicit and to define them precisely. Beyond this goal, the formalism shall be based on a representation that is widely-used in industry such that the learning gap for potential users is relatively small in practice. We also would like to automatically test such role-based policies. For example, we can then check whether rules are missing or contradictory rules exist in the rule set. Hence, a further requirement for selecting a formalism is adequate tool support to carry out this kind of policy validation.

We decided to use UML and its constraint language OCL [Warmer and Kleppe, 2003] for this purpose. This formalism covers the aforementioned requirements quite well. For example, we can employ the USE tool [Gogolla et al., 2007] (UML-Based Specification Environment) for validation purposes. Furthermore, UML is widely used in industry.

The following discussion assumes that the reader is familiar with the core UML concepts, e.g., class and object diagrams, OCL and the USE tool. The papers [Sohr et al., 2008; Kuhlmann et al., 2011; Sohr et al., 2012], which are part of this thesis, describe these concepts in more detail. To obtain deep insight into UML/OCL, the reader is referred to the book by Warmer and Kleppe [Warmer and Kleppe, 2003]. To learn more about USE, one may have a look into its user guide[15].

[15] http://useocl.sourceforge.net/w/index.php/Main_Page

4.2.1 Modeling Static Role-Based Policies according to RBAC96 with UML and OCL

The core concepts for modeling and analyzing with UML/OCL and USE have been described in a paper by Sohr et al. [Sohr et al., 2008]. The main idea for the model introduced in that work was to specify the core RBAC elements, such as the sets *Roles*, *Users*, and *Permissions*, as UML classes and the relations including *UA*, *PA*, and *RH* as UML associations. Authorization constraints (specifically, various forms of static and dynamic SoD properties) have been specified in OCL as OCL constraints. In Figure 13, we give two examples of SoD constraints taken from the IEEE TKDE paper [Sohr et al., 2008]. The first constraint expresses the aforementioned static SoD constraint based on mutually exclusive roles, e.g., between *Cashier* and *Cashier Supervisor*. The second one is a static SoD variant, which prevents conflicting users (e.g., members from the same family) from being assigned to conflicting roles—this rules out situations in which users collude to fool the system. This constraint was taken from the dissertation of Ahn, who has defined a role-based specification language (RCL 2000), which is based on a subset of first-order predicate-logic [Ahn, 1999]. Ahn then specifies various SoD properties within this RCL 2000 formalism including the discussed SSoD-CU constraint.

```
context User
inv SSoD:
let
        CR:Set={{cashier,cashier_supervisor},{r1, r2},...}
in
        CR->forAll(cr|cr->intersection(self.role_)->size()< 2)


context User
inv SSoD-CU:
let
        CU:Set(Set(User))=Set{Set{Frank, Joe},Set{Sue,Lars}},
        CR:Set(Set(Role))=Set{Set{cashier,cashier_supervisor},...}
in
        CR->forAll(cr|cr->intersection(self.role_)->size()< 2)
        and
        CU->forAll(cu|
           CR->forAll(cr|cr->iterate(r:Role|result:Set(User)=Set{}|
               result->union(r.user))->intersection(cu)->size()< 2))
```

Figure 13: Static SoD constraints specified in OCL.

Beyond the specification of role-based authorization constraints with UML and OCL, the TKDE paper discusses validation concepts for role-bases policies. In particular, we showed how conflicting authorization constraints can be systematically detected in role-based policies with the help of the USE tool. In addition, missing constraints could also be identified, i.e., we

presented an approach that lets an analyst check whether an additional authorization constraint––the missing constraint—is not implied by the already defined role-based policy. This is achieved by negating the additional constraint and if we find a system state that satisfies both the policy and the negated constraint, then we can conclude that the additional constraint is actually missing in the role-based policy, i.e., the constraint adds more information to the model.

As we had to use the system state generator ASSL, which is built into the USE tool, for this task and ASSL could only search a relatively small search space, this solution was more a plausibility check rather than an exhaustive search step. Therefore, we later extended this work by employing the model-validator plugin-in of the USE tool, which was not available at that time. Section 4.2.3 discusses how to use the model validator for analyzing role-based policies.

The TKDE paper [Sohr et al., 2008] also outlines how to build an **authorization engine**, a software module that can enforce role-based authorization constraints, based on USE. USE then evaluates requests and decides whether the authorization constraints are satisfied by the request. Later, we have shown how to integrate this authorization engine into a Web service-based environment, i.e., Web service requests are checked against already defined authorization constraints (role-based policies). This authorization system uses the interceptor concept/pattern, i.e., Web service requests are intercepted and then forwarded to the authorization engine, which in turn decides whether access is allowed [Sohr et al., 2008a]. Consequently, the authorization system could be employed by Web service-based applications including banking applications, e-government software and clinical information systems and can enforce different access control policies (please remember that RBAC is policy neutral).

Although useful, the aforementioned RBAC modeling concepts had several shortcomings. This aspect will be discussed in more detail in the following sections.

4.2.2 Modeling Static and Dynamic Role-Based Policies with UML and OCL

The approach outlined in Section 4.2.1 has two shortcomings. First it mixes the level of modeling RBAC concepts with the level of the concrete instances (e.g., roles, users, permissions). Second, it does not support advanced dynamic role-based policies, such as Object-Based Dynamic SoD and History-Based Dynamic SoD, although their relevance has been pointed out

often in literature [Nash and Poland, 1990; Simon and Zurko, 1997; Gligor et al., 1998]. We now discuss both points in more detail.

Let us consider the specification of the conflicting role set `CR` as given in Figure 13:

```
CR: Set = {{cashier,cashier_supervisor},{r1, r2},...}
```

This line directly refers to concrete instances within a specification of a constraint type (SSoD). Then a security officer (administrator) would have to specify SoD rules directly in OCL.

As we cannot expect that administrators in general will learn the OCL formalism, the specification language will probably not be used due to the violated design principle of psychological acceptability. In the TKDE paper, we proposed to resolve this matter by providing a translator between the RCL 2000 language and our UML/OCL-based specification language [Sohr et al., 2008]. In fact, we outlined a straightforward translation between both RBAC specification languages there. Due to constraints on personnel resources, we have not implemented such a tool yet, although this would still be a valuable system for security officers.

Lacking the translation tool mentioned before, a better approach is to separate the specification of the (advanced) RBAC concepts including role hierarchies and the various forms of authorization constraints from the definition of the concrete (instantiated) role-based policies. The latter task is usually done by security officers in organizations, whereas the former task ought to be done by metamodel developers, who are familiar both with UML/OCL and security concepts, i.e., we need an expert in the fields of modeling as well as RBAC. To put it in another way, these experts define the security language for the security administrators/security officers, who in turn use it in their practical work to define role-based policies for their organization. Figure 14 (upper part) visualizes this aspect. The metamodeler is also responsible for introducing and specifying new access control concepts, e.g., new kinds of authorization constraints, such as Permission-Based Dynamic SoD [Gligor et al., 1998] or certain context constraints.

Please also note that we introduce a further participant here, the end user, who uses the application and carries out business/workflow tasks (as part of executing a business process). On executing the business tasks, the application automatically enforces role-based policies as defined by the security officer.

The second shortcoming of the RBAC model from the TKDE paper is the fact only that static and *simple* dynamic SoD constraints (those that use the session concept of RBAC96) can be

expressed. Hence, role-based policies that take the access control history into account, such as Object-Based Dynamic SoD and History-Based Dynamic SoD, cannot be supported. Due to the fact that business processes often need such more advanced dynamic role-based policies, as discussed, for example, by Schaad et al. in the context of a loan originating workflow [Schaad et al., 2006], we decided to extend our UML/OCL specifications accordingly.

In the following, we introduce a UML/OCL-based RBAC specification language, called RBAC-DSL (RBAC Domain-Specific Language), which addresses both shortcomings. The core concepts of this RBAC-DSL have been first described in the work by Kuhlmann et al. [Kuhlmann et al., 2011].

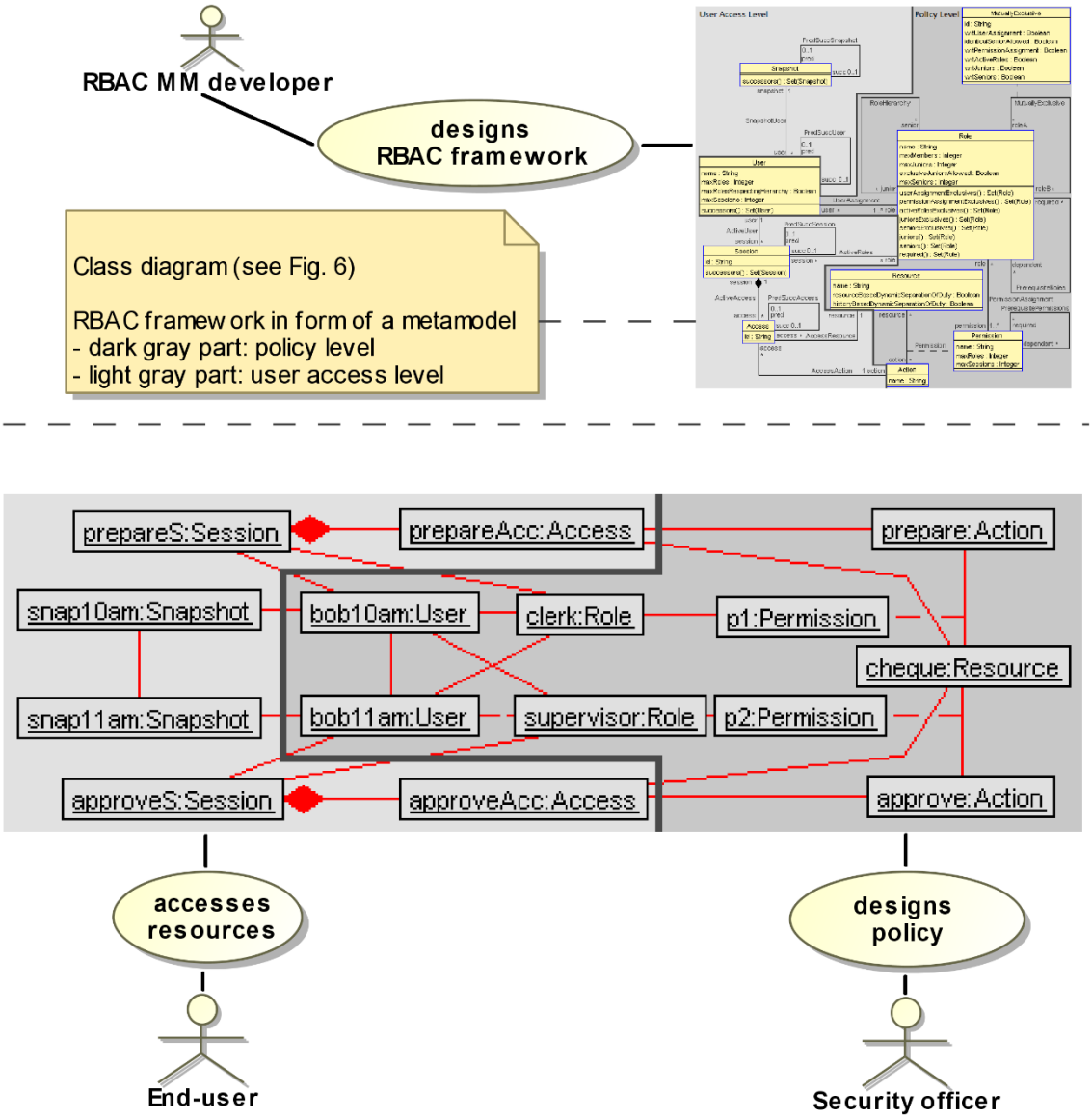

Figure 14: The different roles in an RBAC system: The metamodeler defines the RBAC security language, whereas security officers use this language in their practical work (figure taken from [Sohr et al., 2012]).

Figure 15 depicts the RBAC metamodel, which captures all RBAC96 sets and relations as well as static and dynamic authorization constraints. Please note that we use the term “resource” instead of “object” in this context because otherwise one would mix up the meaning of “object” in the sense of object-oriented modeling with that of access control. For similar reasons, we use the term “action” instead of “operation”.

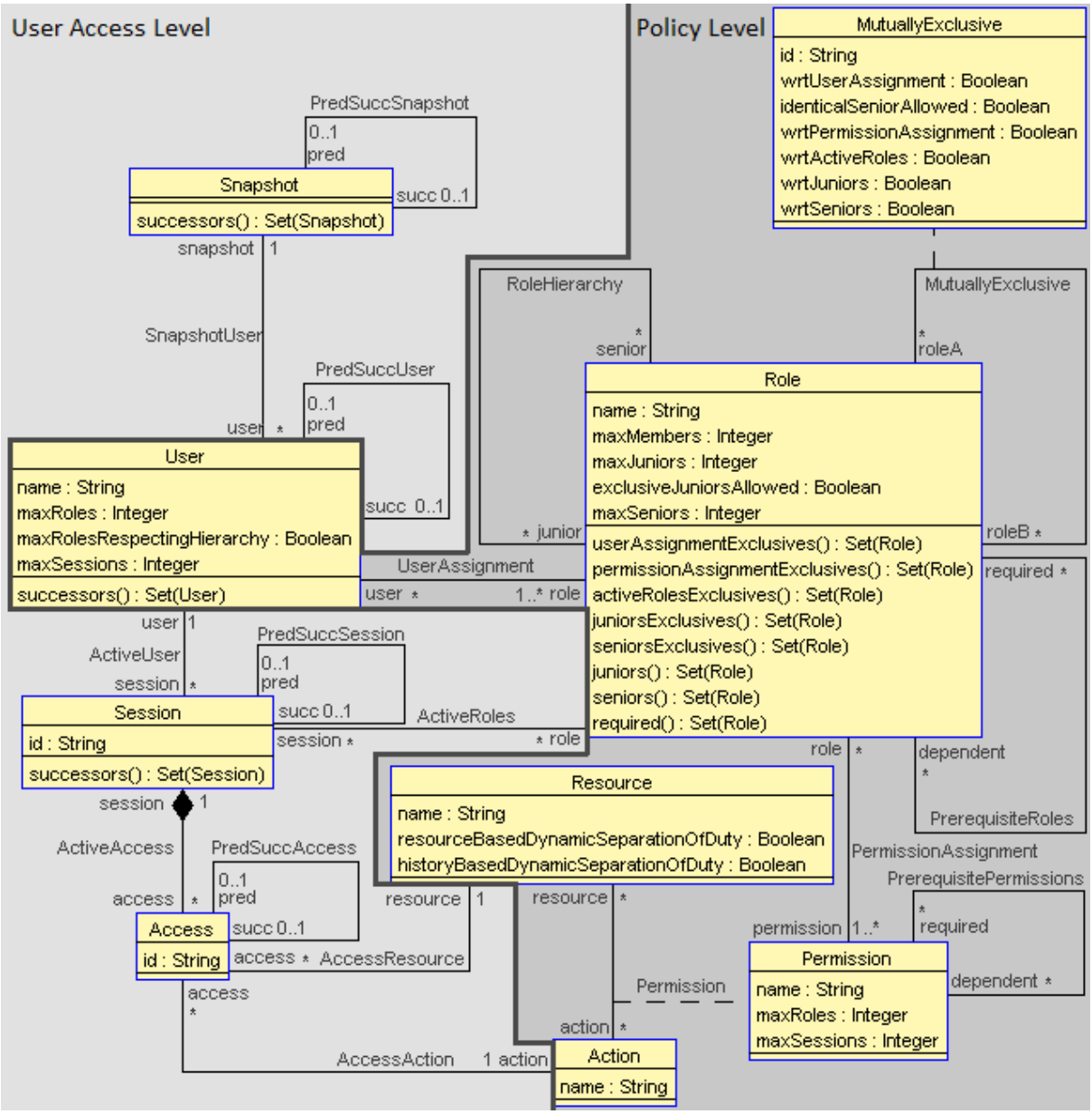

Figure 15: The classes of the metamodel of our RBAC-DSL (taken from [Kuhlmann et al., 2011]).

Each authorization constraint type is expressed by an additional OCL formula. For example, the SSoD constraint is now formulated as follows (using the USE textual syntax for UML classes and class attributes and operations) [Kuhlmann et al., 2013]:

```
context u:User
    inv NoUserAssignedtoExclusiveRoles:
      u.role->forAll(r1,r2|r1.userAssignmentExclusives()->excludes(r2))

userAssignmentExclusives() : Set(Role) =
    mutuallyExclusive[roleA]->select(wrtUserAssignment).roleB->union(
      mutuallyExclusive[roleB]->select(wrtUserAssignment).roleA)->asSet()
```

This OCL constraint (invariant) calculates the set of all mutually exclusive roles of `r1` with the help of the auxiliary operation `userAssignmentExclusives()` and then checks whether `r2` is in that set.

We also extended this RBAC metamodel to support history-based SoD constraints, i.e., we had to introduce a notion of history/time. We refrained from defining an OCL extension with temporal-logic operators like ◇ (sometimes), □ (always) and ○ (next) as proposed by Ziemann and Gogolla [Ziemann and Gogolla, 2003]. This would have meant to provide appropriate tool support for such a temporal OCL extension, which is not available. We rather followed an alternative approach by using the snapshot (filmstrip) model [Gogolla et al., 2014]. This technique duplicates system states and imposes a linear order on them. For this purpose, we defined a `PredSucc*` UML association for each class of the user access level (left part in Figure 15). One can see the resulting effect in Figure 15, e.g., `PredSuccSnapshot` (which determines the successor system state/"snapshot") or `PredSuccSesssion` (which determines the successor session). To impose a linear order on snapshots, supplemental OCL invariants have been necessary. More details can be found in [Kuhlmann et al., 2011; Kuhlmann et al., 2013].

Having extended our RBAC-DSL metamodel with temporal concepts, we can now formulate, for example, Object-Based Dynamic SoD (Resource-Based Dynamic SoD) as an OCL invariant as follows:

```
context u:User
    inv ResourceBasedDynamicSeparationOfDuty:
    Resource.allInstances()->forAll(res |
      res.resourceBasedDynamicSeparationOfDuty.isDefined() and
      res.resourceBasedDynamicSeparationOfDuty implies
      r.access->select(a |
```

```
        u.successors()->including(u)->includes(a.session.user))
                                    .action->asSet()->size() <= 1)
```

Please remember that Resource-Based Dynamic SoD means that a user may perform at most one operation (action) on an object (resource). The access history that is required to express this SoD constraint is introduced by calling `u.successors()`. This auxiliary function collects all representations of the same user `u` over time (i.e., its representation in each system state/snapshot).

Similarly, we can specify History-Based Dynamic SoD as follows:

```
context u:User

  inv HistoryBasedDynamicSeparationOfDuty:
    Resource.allInstances()->forAll(res |
      let availableActions = res.action->size() in
      res.historyBasedDynamicSeparationOfDuty.isDefined and
      res.historyBasedDynamicSeparationOfDuty and
      availableActions > 1 implies
        res.access->select(a |
          u.successors()->including(u)->includes(a.session.user))
            .action->asSet->size() < availableActions)
```

This specification differs from the former one in that a user must not perform all operations (actions) against an object (resource). This fact is expressed by a check against the `availableActions` local variable. As our description is here somewhat telegraphic, we again refer the reader to the MSCS journal paper [Kuhlmann et al., 2013], which discusses these specifications in more detail.

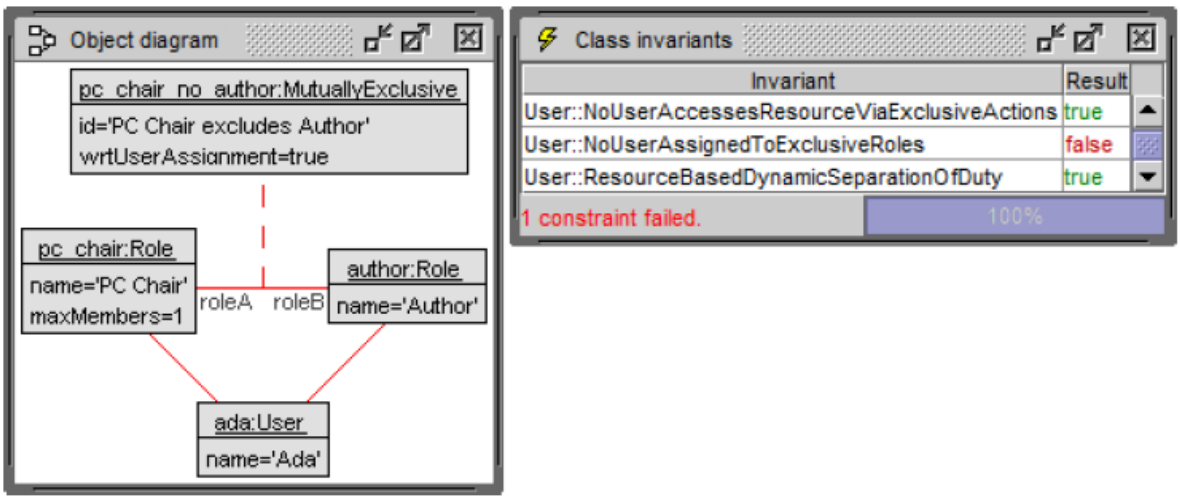

Figure 16: An example static SoD policy including a test case (figure taken from [Kuhlmann et al., 2013]).

We have seen that the OCL specifications of authorization constraints can become quite complex and are hence difficult to understand. Therefore, it becomes clear that only specialists (metamodelers) can do the task of defining new types of authorization constraints in OCL. A security officer or administrator, however, is usually not confronted with these OCL specifications. She defines role-based policies at the instance rather than the metamodel level. In the left part of Figure 16, we show an example of a role-based policy, which is defined at the instance level in form of a UML object diagram. One can see that the roles *Pc_chair* and *Author*[16] have been defined as mutually exclusive, i.e., a link between both roles exists, which is an instance of the UML association class `MutuallyExclusive`. Please note that the administrator need not know about the concrete OCL constraint `NoUserAssignedtoExclusiveRoles` (which represents static SoD in terms of mutually exclusive roles); she only needs to know about the semantics of the constraint type static SoD, i.e., she must possess adequate knowledge on role-based security concepts.

Similarly to static SoD, a security officer can specify instances of other authorization constraint types, e.g. History-Based Dynamic SoD. On the right-hand side of Figure 17, for example, one can see that a History-Based Dynamic SoD constraint is set for the `resource2`. An administrator has done this by having set the attribute `historyBasedDynamicSoD` to true. The other parts of the policy defined in Figure 17 comprise a UA relation between `role2` and `user2` as well as PA relations between `permission1` and `permission2` with `role2`.

[16] The example comes from a case study, in which role-based policies are modeled for the conference system EasyChair.

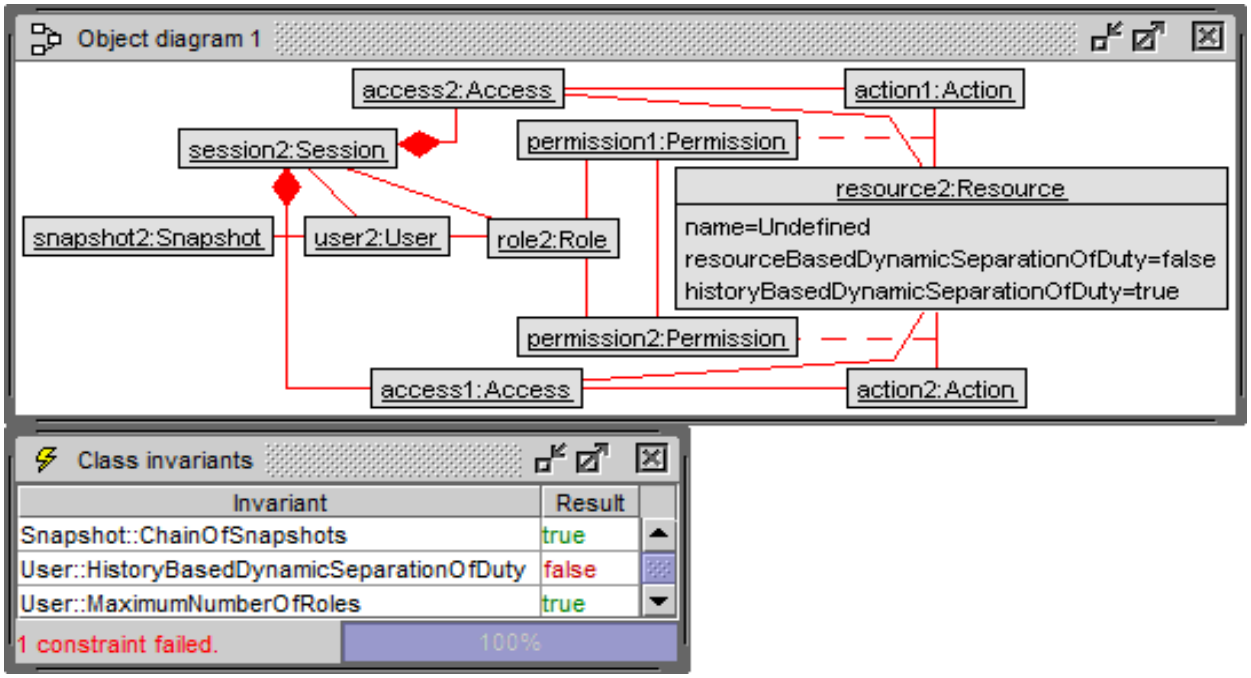

Figure 17: A test case for History-Based Dynamic SoD (figure taken from [Kuhlmann et al., 2013]).

4.2.3 Testing Role-Based Policies

One further benefit of using our modeling approach is the ability to test role-based policies. Then a security officer, for example, can check whether authorization constraints are missing, i.e., the defined role-based policy is too permissive. Also, desirable scenarios (test cases/system states) may be inadvertently ruled out by the policy—in this case, the defined policy is too strong. The core concepts of validating role-based policies have been introduced in two works [Sohr et al., 2008; Kuhlmann et al., 2011], but we also give a high-level overview of these aspects in the following paragraphs.

First one can define concrete positive and negative test cases. Positive test cases are UML object diagrams that are expected to be satisfied by the defined policy. If this test case is not fulfilled, then either the policy or the test case is wrong. Negative test cases describe situations (represented as object diagrams) that must not be satisfied by the defined role-based policy. If so, then again either the policy or the test case is wrong. This kind of validation can be performed with a UML validation tool, such as USE, which supports most OCL concepts [Sohr et al., 2008; Kuhlmann et al., 2011].

Later versions of the USE tool, support a model validator plugin [Hilken et al., 2014]. This plugin allows an analyst to automatically generate complex test cases. In particular, this technique is interesting when dynamic SoD policies are to be considered. To check dynamic (SoD)

policies, strictly speaking, one needs to execute the application or system in question. The model validator, however, automatically generates test cases that encompass execution passes of the application or system similarly to a simulation. Figure 17 depicts such a situation. One can see that user `user2` accesses `resource2` via session `session2`. In this case, `user2` performs both `action1` and `action2` on `resource2`, which violates the defined History-Based SoD constraint (see also the lower part of Figure 17). The model validator can also generate more complex test scenarios including several points in time (snapshots)—in this example, however, only one snapshot is considered.

Another point should be regarded with this validation approach. Due to memory restrictions, the search space of the model validator must be limited. If the model validator finds a test case that violates the policy, then we know that the policy is wrong. However, if the model validator does not find such a scenario, it could also be the situation that the search space was too small to detect such a test case. In this case, one cannot be sure whether the policy is correct or not.

4.2.4 Modeling Role-Based Delegation with UML and OCL

One further advanced access control concept is **delegation**. Delegation is an important factor to satisfy dynamic requirements for secure distributed computing environments. Several definitions of delegation exist in literature [Barka and Sandhu, 2000; Zhang et al., 2003; Gladney, 1997; Wainer and Kumar, 2005; Sohr et al., 2008]. Delegation generally refers to an activity in which one active entity in a system delegates its authority to another entity to carry out some functions on behalf of the former [Sohr et al., 2008].

Role-based delegation allows a principal to delegate *roles* to other principals rather than delegating permissions. Role-based delegation is a useful access control concept in organizations, in particular in hospitals, financial institutes, or governmental agencies [Sohr et al., 2005; Sohr et al., 2012]. In a hospital, for example, a treating physician may (temporarily) give access rights to electronic health records to a specialist in certain situations (e.g., through a *Read patient data* role). Another example of delegation in a hospital setting is a situation in which a patient is transferred from one ward to another one.

We extended our RBAC-DSL to support role-based delegation concepts, which allows us to also express role-based policies for clinical information systems or governmental applications [Sohr et al., 2012]. We decided to use the RDM2000 model [Zhang et al., 2003] for this purpose

because this model has been sufficiently formally specified and it is based upon the RBAC96 model, which is the foundation of our RBAC-DSL.

We do not give the detailed concepts of RDM2000 here because this model has been described in another work [Sohr et al., 2012], which is also part of this cumulative thesis. We only enumerate the main concepts briefly in the following.

RDM2000 extends the *UA* relation with two further relations *UAO* (original user assignment) and *UAD* (delegated user assignment) with the condition $UA = UAO \cup UAD$. This means that the delegated user assignment and the user assignment as defined in RBAC96 form the complete *UA* relation. Three further relations have been introduced:

- *DLGT* is a delegation relation. It states that user *u* with role *r* delegates role *r'* to user *u'*.
- *ODLGT* is an original user delegation relation.
- *DDLGT* is a delegated user delegation relation.

Finally, the condition $DLGT = ODLGT \cup DDLGT$ holds. Please note that *DDLGT* allows for cascading delegation, i.e., user *u* may delegate a role *r'* to a user *u'*, who in turn delegates that role to a user *u''*.

RDM2000 allows one to impose restrictions on who is allowed to delegate role *r* with the help of the relation *can_delegate*. In essence, this relation represents authorization constraints with respect to delegation. Typically, constraints include the maximum delegation depth or roles that the delegated user must be a member of (prerequisite role).

Furthermore, RDM2000 supports the concept of **revocation**. Revocation allows the delegating user *u* to revoke a role *r* from the delegated user *r'*. Analogously, a *can_revoke* relation has been introduced. For example, one can demand via *can_revoke* that only the user *u* who has delegated the role *r* can later revoke it (grant dependency).

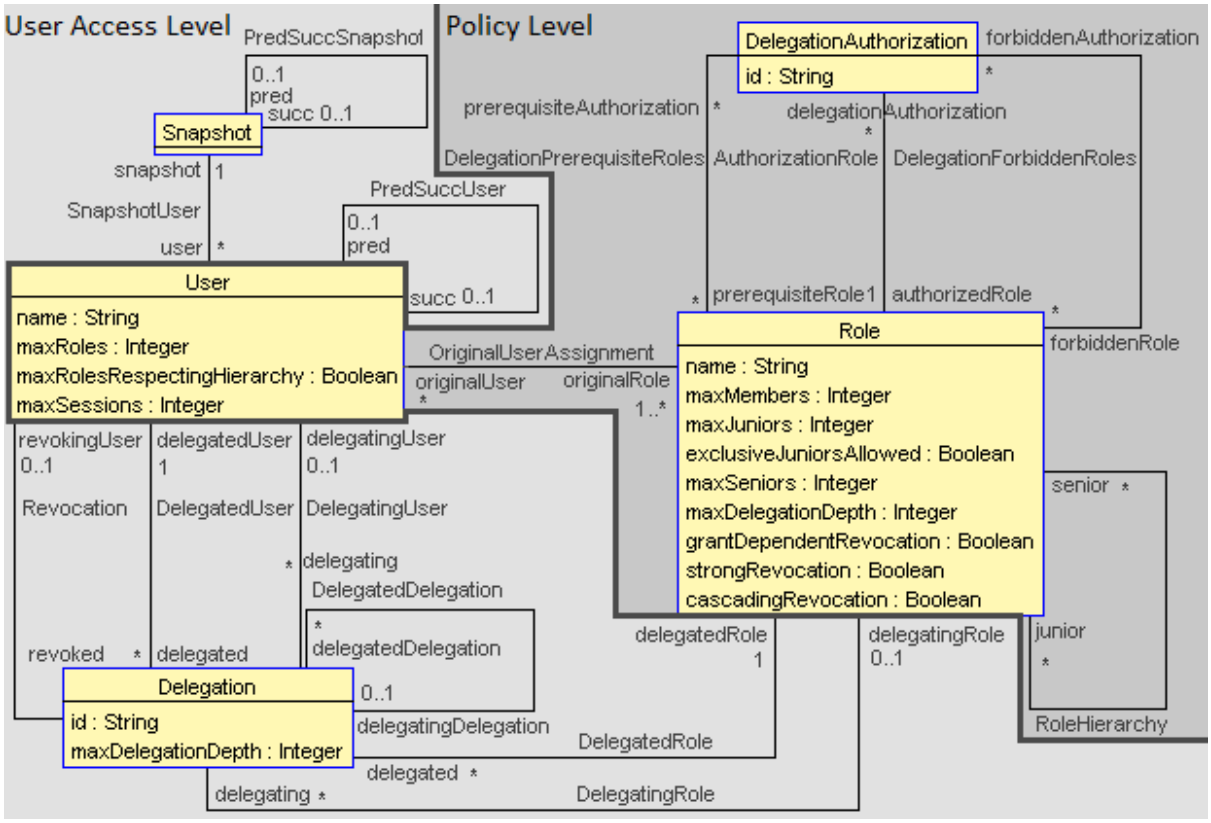

Figure 18: An extended RBAC metamodel supporting delegation and revocation concepts (figure taken from [Sohr et al, 2012]).

Figure 18 shows the metamodel, which extends the RBAC counterpart depicted in Figure 15 with delegation and revocation concepts as defined in the RDM2000 model. Please note that a series of OCL constraints also exist that formalize the RDM2000 model and which can be found in another work [Sohr et al., 2012]. One can see in Figure 18 that the two new classes `Delegation` and `DelegationAuthorization` have been introduced. The former describes the *ODLGT* and *DDLGT* relations, whereas the latter is, e.g., used to express authorization constraints on the delegation step, such as prerequisite role constraints. Please further notice that additional attributes have been defined for the `Role` class, such as `maxDelegationDepth` and `grantDependentRevocation`. These attributes are then accessed within OCL constraints.

A security officer can now define delegation and revocation policies, which are based on this metamodel, as UML object diagrams. Furthermore, she can then employ the USE tool to test the policy and to automatically generate test cases with the help of model-validator plugin [Sohr et al., 2012].

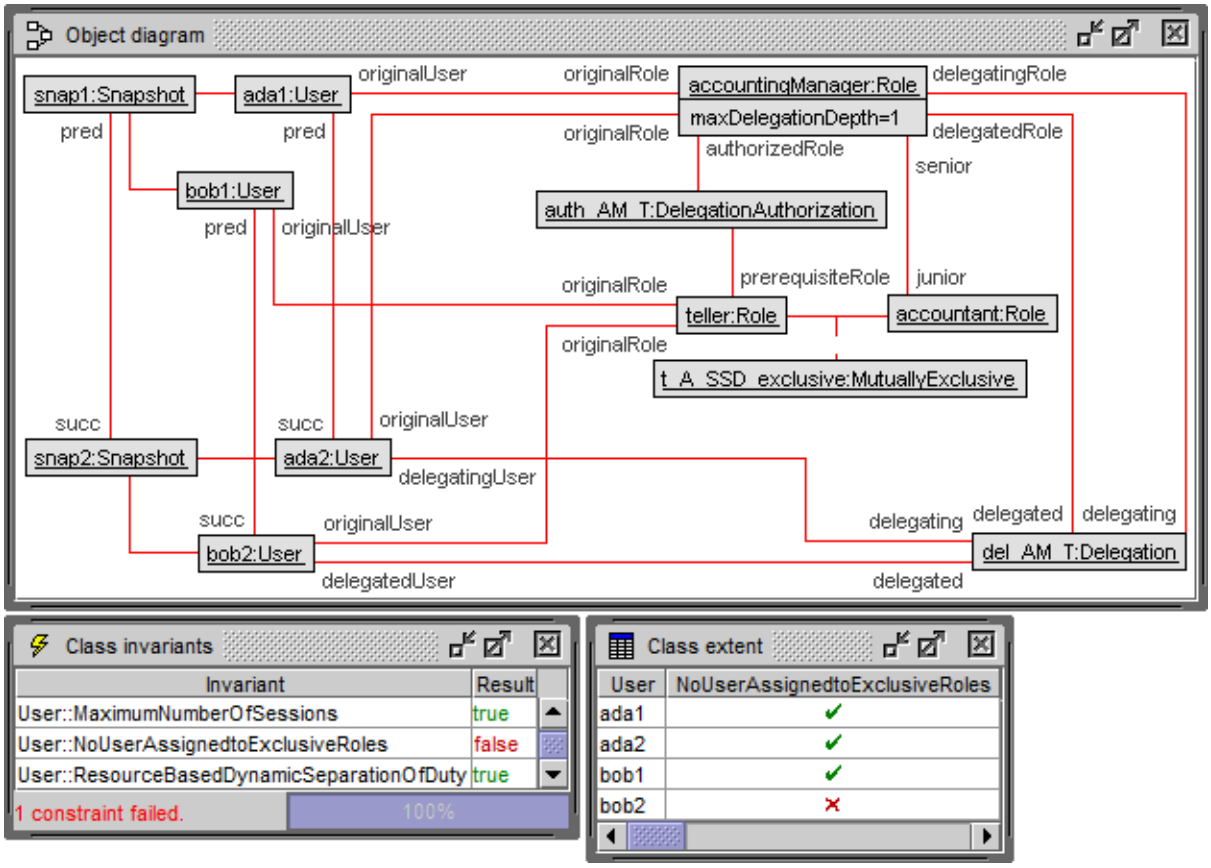

Figure 19: A test scenario that violates static SoD after a delegation step (figure taken from [Sohr et al., 2012]).

In Figure 19, for example, we show a test scenario in which delegation rules conflict with a static SoD property. User `Ada` delegates the role `AccountingManager` to user `Bob`. Furthermore, a static SoD constraint between the roles `Accountant` and `Teller` exists as well as a role hierarchy relation between the roles `AccountingManager` and `Accountant`. We further assume delegation constraints, which state that the delegated user must be a member of the `Teller` role and that the maximum delegation depth is 1 (no further delegation is allowed). The scenario depicted in Figure 19 violates this policy by allowing user `Bob` to be a member of the `Accountant` and `Teller` roles. `Bob` has obtained the role `Accountant` indirectly through the role hierarchy after he has obtained the `AccountingManager` role through delegation. As he already possesses the `Teller` role, static SoD is violated, i.e. delegation and SoD rules may conflict in certain cases.

4.3 Summary and Outlook of this Work

Despite the aforementioned limitations of our approach, we believe that it helps security officers more systematically manage role-based policies. In particular, the validation step supports a security officer in testing role-based policies without being an expert in formal methods. A methodology and tools for specifying and testing make the management of role-based policies more transparent; role-based policies can finally be defined in a revision-proof manner.

In follow-up research we plan to apply our modeling and validation approach to real-world applications. For example, we can study clinical information system policies in this context. Such a case study would be of particular interest as it comprises different advanced role-based concepts including various forms of SoD, context constraints, different forms of delegation, as well as role hierarchies. One current obstacle to carry out this research is the fact that vendors of clinical information systems often have implemented their own role-based mechanisms without systematically having in mind the advanced role-based concepts mentioned in this chapter. Often role-based concepts have been integrated into these applications in an ad hoc fashion. Furthermore, vendors of clinical information systems are often not willing to share information about security and privacy mechanisms with external partners.

The application in the area of clinical information systems also demonstrates that our role-based modeling approach can be used within a privacy-by-design development methodology [Danezis et al., 2015]. Role-based policies allow a privacy architect to formulate privacy requirements as authorization constraints and validate them with the help of tools like USE.

Due to our contacts to a large enterprise, however, we have an early indicator that the methods introduced in this chapter are actually needed in industry. In fact, this industrial setting is similar to that of clinical information systems: In both cases, role-based mechanisms have been implemented in applications without specifying or at least documenting these concepts. Chapter 5 will discuss this topic in more detail by means of a case study with a real-world-application.

More and technically deeper information about the current chapter's topic can be found in different publications [Sohr et al., 2008; Kuhlmann et al., 2011; Sohr et al., 2012; Kuhlmann et al., 2013]. The journal paper [Sohr et al., 2008] lays the foundation, whereas the other publications concretize the ideas and improve the UML metamodel for RBAC and its advanced concepts. All these publications are part of this thesis and can be found in the appendix.

Chapter 5

Extracting and Analyzing Role-Based Policies from a Real-World Java Business Application

This chapter combines techniques described in the previous two chapters into a unified approach for managing role-based policies in real-world Java applications. The basic idea is to extract the implemented role-based policies from the application code and transform them into UML object diagrams. These diagrams, in turn, can then be analyzed by a UML validation tool [Gogolla et al., 2007; Przigoda et al., 2015].

The background of this work is a concrete practical problem of a large enterprise[17]. The central security department, also responsible for managing enterprise-wide access control, intends to switch to an access management based on RBAC. One problem of this process, however, is that the central security department does not know in detail, which role-based policy is actually implemented in which application. As this enterprise runs hundreds of applications, which in total implement several thousands of business processes, a methodology must be developed to make explicit these implemented policies and transform them into a form amenable to validation. Furthermore, it should be possible to combine role-based policies of the several applications into one *common policy* such that access to and within applications can be controlled centrally.

Our approach to addressing the aforementioned problem has been initially applied to a business application, in the following called "B2BApplication", which the enterprise uses to manage all their suppliers. Although the project was set up as a feasibility study, the B2BApplication was of moderate complexity with more than 125,000 lines of code. The goal of this feasibility study was to develop a methodology that the enterprise can apply to various applications over the next years. This methodology should finally allow the enterprise to establish the envisioned central role management of their applications.

[17] The name of the enterprise cannot be given due to non-disclosure agreements.

In the following discussion, we use the term “low-level model” to describe the role-based policy that is implemented in the B2BApplication, whereas the term “high-level model” refers to the intended and consolidated role-based policy. The low-level model is extracted from the Java source code of the B2BApplication and additional information stored in the underlying database. We use these terms as they have been introduced within our discussions with the industrial partner.

In terms of authorization concepts, the low-level model corresponds to the Policy Enforcement Point (PEP). The PEP consults a Policy Decision Point (PDP) for access decisions. These components working together create a standard authorization framework as introduced in the OASIS specification [Organization for the Advancement of Structured Information Standards, 2013] and establish the enforcement environment of an RBAC system. In our terms, the PDP correlates to the “high-level” model (policy), which the implementation should enforce.

We follow the modeling approach introduced in Section 4.2. This means that the high-level model (policy) is represented as an instance model of a UML-based metamodel. This metamodel, however, had to be adjusted and extended to capture the enterprise’s specific access control concepts (see Section 5.1).

The low-level model is also represented as an instance of a metamodel. Due to the fact that the roles and permissions used in the B2BApplication are stored in the application’s database, the data model of that database (essentially the database scheme) can be utilized to automatically obtain a metamodel for the low-level policy. This aspect will be discussed below in more detail.

The remainder of this chapter explains the single steps of our methodology to manage and validate the implemented role-based policies. Thereafter, we summarize the current status of our approach. We also reflect upon open points, which partly stem from the fact that the software was not developed with having such a security analysis mind. Therefore, we briefly discuss how applications should be structured to support our analysis process.

5.1 The Metamodel for Role-Based Access Control of the Business Application

We only show excerpts of the metamodel for RBAC here. The details are given in a further report [Berger, 2016]. As indicated above, we follow the approach presented in Section 4.2. In particular, we support RBAC96 concepts as well as simple authorization constraints, such as

static SoD, cardinality, and prerequisite role constraints. We also consider some specific access control concepts that are used by the enterprise. For example, in the B2BApplication users can obtain permissions directly without having assumed any role[18]. Another difference is that a user may have different logins with different permissions/roles. The large enterprise uses the concept of logins to express the user's possibility to access different applications with different logins, e.g., one login for SAP business software and another login for a CAD application.

We use Ecore [Steinberg et al., 2009] rather than UML as the modeling language for defining the RBAC metamodel. As Ecore resembles in its concepts UML, we expect that a translation between both modeling languages should be easily possible. We have chosen Ecore for two reasons. First, it is integrated into the Eclipse IDE, which allows a security analyst or developer to use it directly during the software-development process. Second, Ecore models are supported by the DFKI OCL validation tool [Przigoda et al., 2015][19], which was used within this project.

Figure 20 to Figure 23 show the Ecore classes that are part of the metamodel for the access control concepts required by the B2BApplication. Due to the fact that the metamodel comprises general access control/RBAC concepts, it can be applied to other applications as well.

In Figure 20, the high-level concepts for accessing (executing) the system have been defined. A subject (e.g., a user) executes operations and produces access traces this way. The access control concepts are represented by the `Constraint` class.

[18] This supports the hypothesis that the role-based access control concepts that are implemented in many applications do not systematically follow the ANSI RBAC standard [American National Standards Institute, 2004]. Often this standard is not even known to developers.

[19] This OCL validation tool was developed at the German Research Center for Artificial Intelligence (DFKI), Bremen.

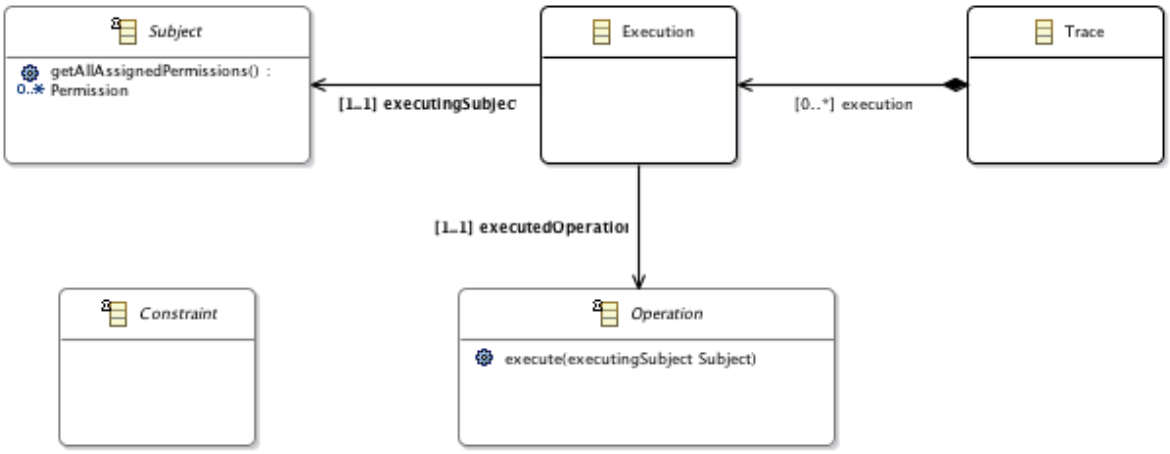

Figure 20: High-level overview of system execution modeled in Ecore (figure taken from [Berger, 2016]).

Figure 21 depicts the Ecore classes that represent the general access control concepts supported in our metamodel. In particular, the `Constraint` class generally expresses constraints on executing operations. Constraints include permissions, but also roles and role-based authorization constraints. Please note that we use the generalization concept provided by UML/Ecore to express this fact. The `Constraint` class generalizes the `Permission` class (see Figure 21), the `Role` class (see Figure 22) as well as classes representing authorization constraints (e.g., the `RoleExclusion` class in Figure 22).

Figure 21 also shows the two additional classes `EClassifier` and `EStrcuturalFeature`, which are predefined in the Ecore metamodel. The former comprises classes, whereas the latter represents class attributes. We use both these Ecore metamodel classes because we also intend to represent domain-specific data models. For example, instances of `EClassifier` are the concrete classes of B2BApplication's data model, such as `ums_contract` (contract with a business partner) or `ums_policy_agreement` (information about the fact that the partner has agreed with the enterprise's policy). Instances of `EStructuralFeature`, however, are attributes of those classes, e.g., `preferredLanguage` or `validUntil` for the `ums_contract` class. As our metamodel is general, one can define data models for other applications as well. For example, in a healthcare setting, a class `electronic_health_record` with an attribute `drugPrescriptions` are conceivable.

Access can then be restricted both at the level of entire classes and single class attributes. This fact is modeled with the help of two additional associations, one between the classes `DataAccess` and `EClassifier` and the other one between `DataAccess` and `EStructuralFeature`.

Figure 22 describes the metamodel classes for RBAC. The core class of this package is `Role`, which is a subclass of `Permission`. Beyond the notion of role, we also model classes for expressing authorization constraints—for each kind of authorization constraint, we define a separate class, e.g., `RoleExclusion` for static SoD or `PrerequisiteRole` for prerequisite role constraints. Please compare this modeling approach with that of the RBAC-DSL (see also Section 4.2.2). There, we had to add attributes, such as `maxMembers`, to the `Role` class to express cardinality constraints. From the viewpoint of software modeling, however, this approach is not ideal as it bloats the `Role` class with attributes, which are only needed in certain cases—when a cardinality constraint is used within an organization. Hence we decided to introduce further classes for authorization constraints, which can be added to the metamodel as needed.

Similarly to the RBAC-DSL metamodel introduced in Section 4.2, we also define accompanying OCL constraints. In the following, we give two examples of such OCL specifications:

```
context RoleExclusionCheck:

inv atMostOneExclusiveRole:
  check.excludingRoles->selectByKind(permissions::Permission)
                      ->intersection(person->collect(logins)
                      ->collect(assignedPermissions))->size() = 1;
```

```
context PrerequisiteRoleCheck:

inv allRequiredRolesIncluded:
  let assignedPermissions : Set(highlevel::permissions::Permission)
     = person.getAllAssignedPermissions()
  in if assignedPermissions->includes(check.role) then
       assignedPermissions->includesAll(check.requiredRoles)
     else
       true
     endif;
```

The former OCL specification formalizes static SoD, whereas the latter describes the prerequisite roles authorization constraint. The OCL constraint for static SoD calculates the set of all roles of a person (user) who is registered with the application. This set is compared to the

excluding role set `excludingRoles`. Please remember that a person may have different logins, i.e., the roles must be collected over each login. We use an auxiliary class `RoleExclusionCheck` (not shown in Figure 22), which connects the static SoD rule with a concrete instance of the `Person` class. This auxiliary class has been introduced for diagnosis purposes because now an analyst can directly see in a UML validation tool's output which persons violate the rule.

The OCL invariant for the prerequisite role constraint assures that the person is assigned to all roles that are in the `check.requiredRoles` set. Again we have introduced a helper class for diagnosis purposes (not shown).

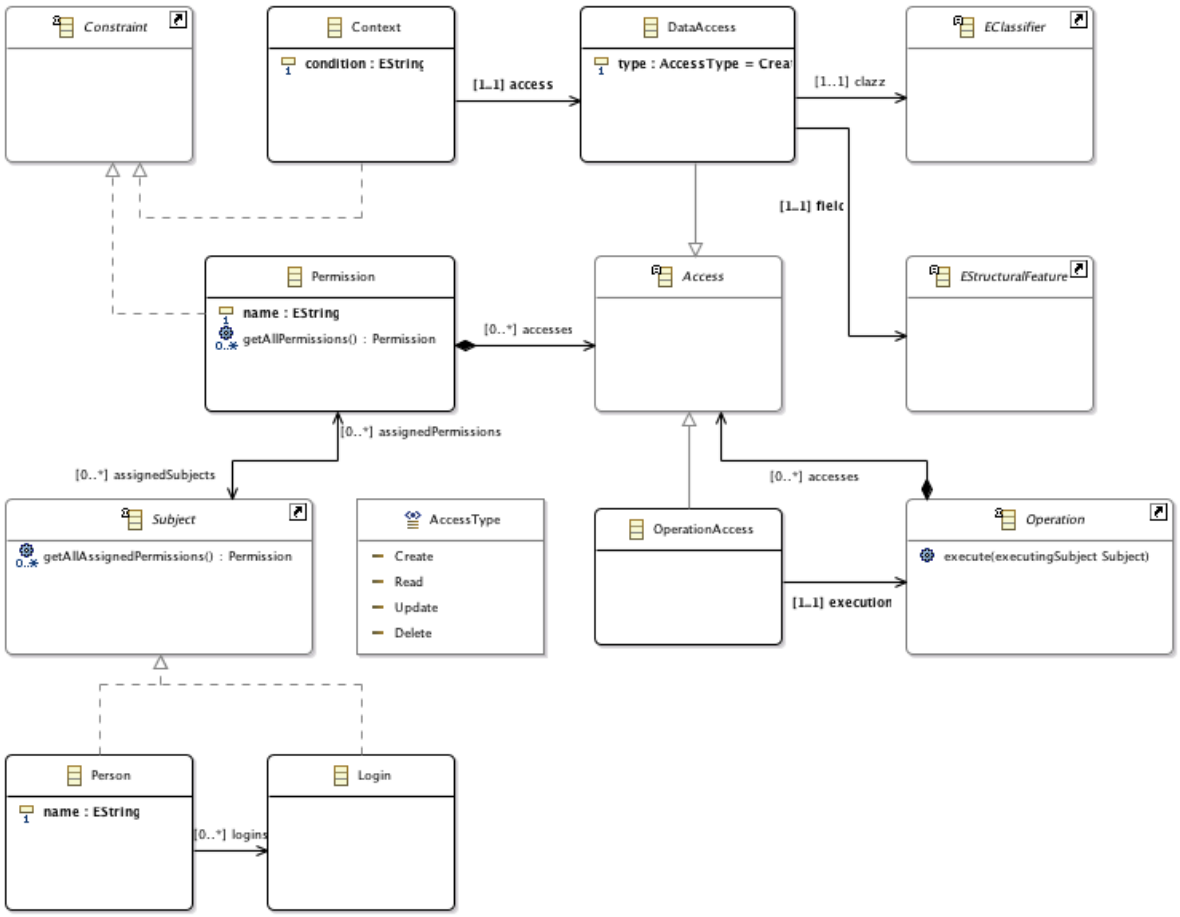

Figure 21: Ecore model part for general access control concepts of the B2BApplication (figure taken from [Berger, 2016]).

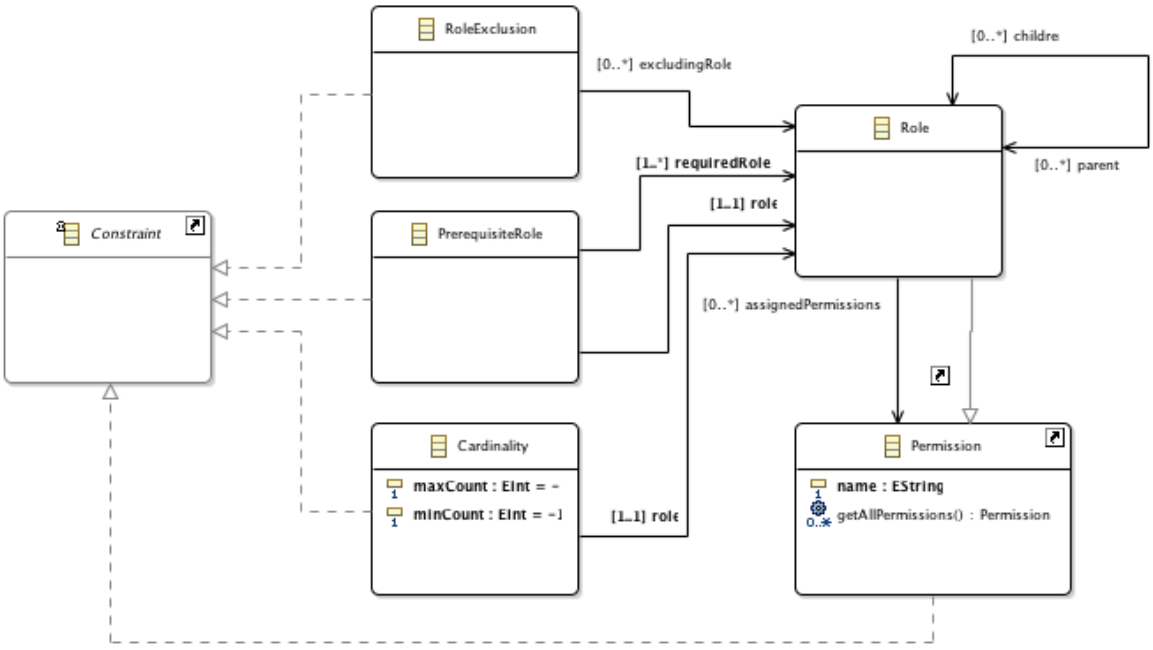

Figure 22: Ecore model part describing advanced RBAC concepts (taken from [Berger, 2016]).

Beyond role-based concepts, our metamodel comprises classes that describe business processes (workflows). The background is here that the enterprise makes available several business process descriptions, which the B2BApplication ought to have implemented. Consequently, information about business processes has been integrated into the high-level model for role-based policies. Single steps (tasks) of these processes can then be protected by certain roles. Figure 23 depicts the part of the metamodel, in which the notions of processes, steps, and masks have been introduced. The `Process` class describes a business process, while `Step` represents a single process task. The masks are required because business process steps are often implemented by masks (user interfaces) in applications, which is particularly the case in the B2BApplication.

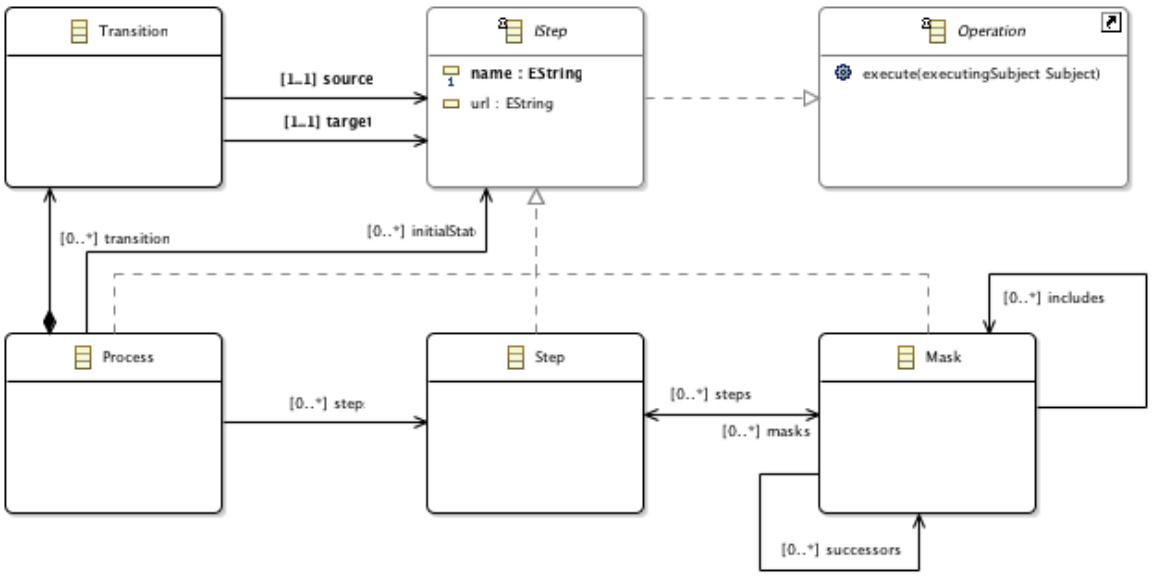

Figure 23: The Ecore model part for describing workflows/business processes (taken from [Berger, 2016]).

5.2 The Single Steps of our Analysis Approach

Having introduced the core concepts of our RBAC metamodel tailored to the needs of the B2BApplication, we now present the single steps of our analysis approach. Figure 24 shows these single steps and their execution order.

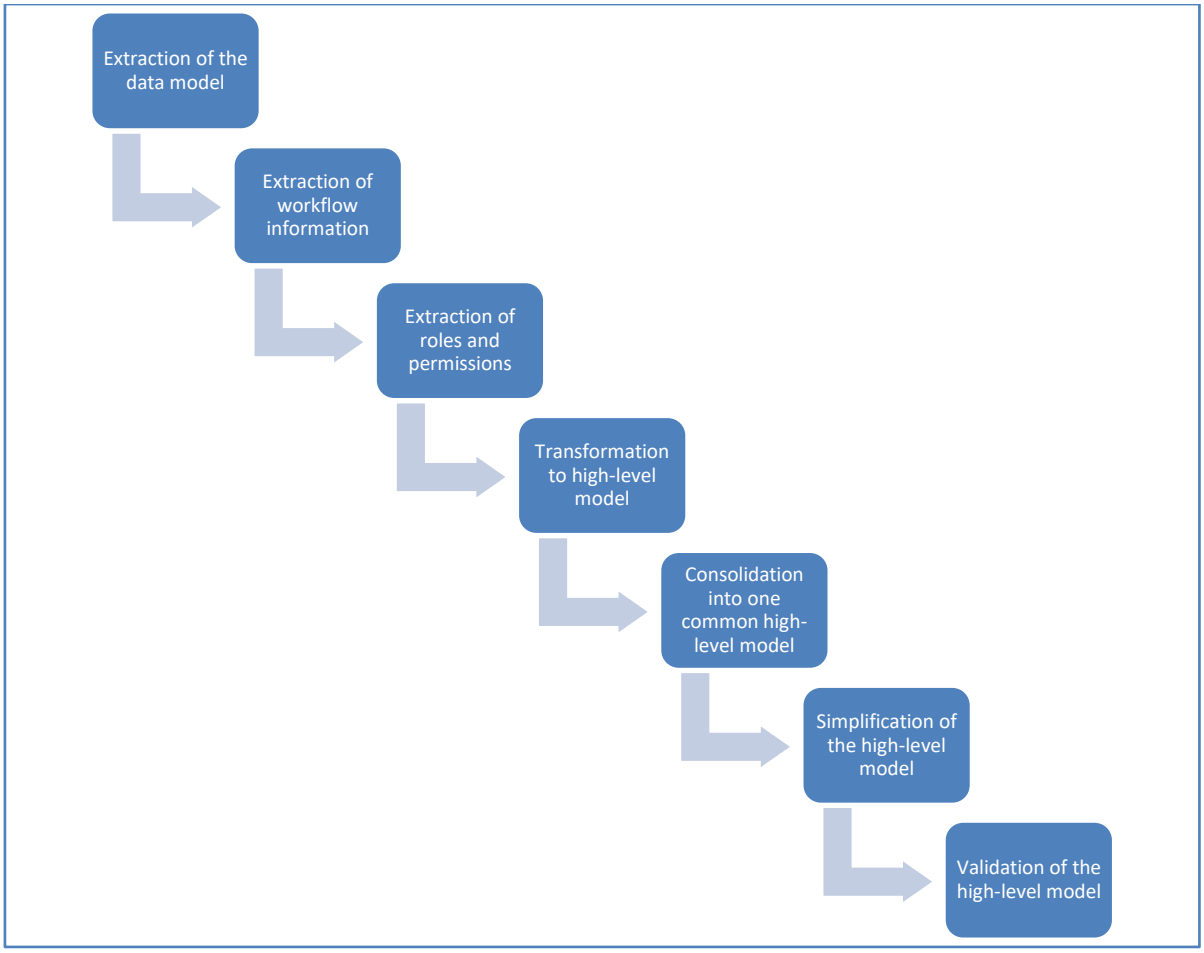

Figure 24: The single steps of the analysis approach to validating the business application.

Extraction of the of domain data model, workflow information as well as roles and permissions

We use static program analysis techniques to extract different information from the Java source code of the B2Bapplication and employ the Soot tool [Vallée-Rai et al., 1999] for this purpose. As indicated above, the B2Bapplication stores roles and permissions in a database. We extract the database scheme, which in turn is used to define the metamodel for the low-level (instance) model. In particular, we evaluate Java Persistence API annotations, including `@Entity`, `@Embeddable`, `@Table`, `@Column`, `@ManyToOne`, `@OneToMany`, and `@Transient` for this purpose. We utilize type information contained in the Java bytecode to extract the persistence annotations and add this information to the (low-level) Ecore model. Typical classes of this regained model are `ums_login`, `ums_profile`, `ums_person`, `ums_contract`, `ums_permission`, and `ums_role`. They describe access control concepts that are used in

the B2BApplication, such as login, profile, person, contract, permission, and role. An excerpt of the extracted metamodel can be found in Figure 25.

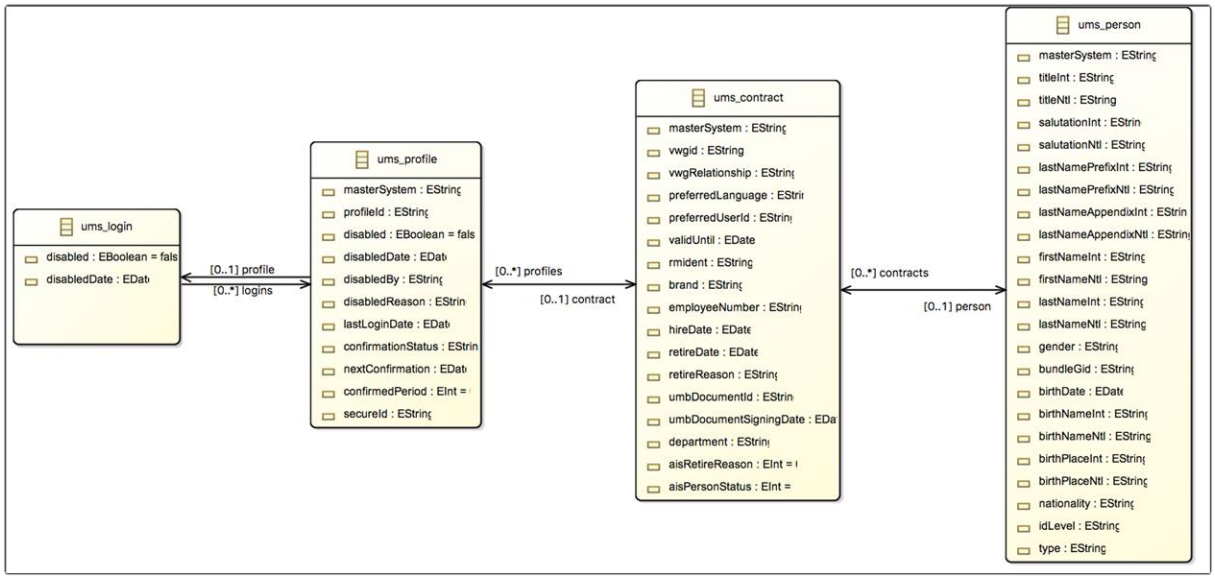

Figure 25: Excerpt of the low-level metamodel (figure taken from [Berger, 2016]).

After reconstructing parts of the metamodel, we extract concrete instances of these classes from the source code (bytecode) by employing static analysis techniques by means of the Soot tool. In particular, we extract concrete roles and permissions by searching for specific authorization API calls, which are used in the B2BApplication's implementation. In particular, we search for authorization API calls, such as

- `de.enterprise.ums.web.permissions.Permissions: boolean hasPermission (java.lang.String,java.lang.Object[])`
- `de.enterprise.ums.info.permission.PermissionEvaluator: boolean hasPermission(java.lang.String,java.lang.String)`
- `de.enterprise.ums.admin.server.logic.ProfileConfirmationLogic: boolean hasPermission(java.lang.String,java.lang.String, de.enterprise.ums.persistence.IPersistenceTransaction)`
- `de.enterprise.ums.admin.server.logic.ProfileLogic: boolean hasPermission(java.lang.String,java.lang.String, de. enterprise.ums.persistence.IPersistenceTransaction)`
- `de.enterprise.ums.info.internal.PermissionEvaluatorImpl: boolean hasPermission(java.lang.String,java.lang.String).`

String parameters of these API calls stand for permission, profile, and role parameters. The code uses naming conventions for differentiating between these types, e.g., beginning the string parameter with `role_` means that we have encountered a role. Again Soot helps us identify the API calls listed above and extract their string parameters.

The third type of information that we obtain from the code is the sequence of masks the application's user interface consists of. We later utilize this information to compare the implemented process steps (the masks sequence) with the defined business process to detect inconsistencies between the design and the implementation. Technically, we partially reconstruct workflow information (the mask sequences) by scanning the available Java Server Faces[20] (JSF) servlets and the Seam configuration. Seam[21] is the Java-based web framework and JSF is a software framework for developing graphical interfaces for Web applications. We transform each JSF servlet into an instance of the `Mask` class. Then we use `include` relations to identify the call order of the masks. We also obtain further information from the Seam configuration about mask transitions. The extracted mask information can then be manually connected to task instances, which are obtained from business process descriptions (as part of the high-level model), with the help of an Eclipse editor.

Model-to-Model transformation

This step automatically converts the low-level instance models to a high-level instance model. In particular, we define a model-to-model transformation based on Eclipse QVTo, a model-transformation engine, provided by Eclipse. QVTo (QVT operational) is a language for semi-formally specifying model-to-model transformations [Object Management Group, 2007]. The transformation is defined at the meta-model level.

Figure 26 shows an excerpt of the QVTo transformation definitions. The three low-level classes `ums_runtime_role`, `ums_login`, `ums_person` are mapped to the high-level classes `Role`, `Login`, and `Person`, respectively.

The given specifications are quite simple and transform the low-level entities into their high-level counterparts. The transformation specification, however, does not only define transformations between class instances, but also between association instances. For example, the

[20] http://www.oracle.com/technetwork/java/javaee/javaserverfaces-139869.html

[21] http://seamframework.org/

transformation `toPerson()` collects all `ums_login` instances that are (indirectly) assigned to a `ums_person` instance and inserts corresponding links into the high-level instance model. Please note that in the high-level instance model `Login` instances are directly assigned to `Person` instances, i.e., no counterparts to the `ums_contract` and `ums_profile` classes exist here anymore. This effect is shown in Figure 27 and Figure 28, respectively. For example, the person (user) `Karsten Sohr` is connected with `Login1-KS` and `Login2-KS` via `Contract-KS` and `Profile-KS` in the low-level instance model. In the high-level instance model, however, `Karsten Sohr` is directly connected to both logins.

```
   transformation LowLevelToHighLevel(in input : ll, out output : hl);
   main() {
    input.rootObjects()[ums_runtime_role]->map toRole();
    input.rootObjects()[ums_login]->map toLogin();
    input.rootObjects()[ums_person]->map toPerson();
   }
   mapping ums_login::toLogin() : hl::permissions::Login {
    assignedRoles := self.profile.runtimeRoles->resolve(hl::rbac::Role);
   }
   mapping ums_person::toPerson() : hl::permissions::Person {
    name := self.firstNameInt + " " + self.lastNameInt;
    logins := self.contracts->collect(profiles)->collect(logins)
                                    ->resolve(hl::permissions::Login);
   }
   mapping ums_runtime_role::toRole() : hl::rbac::Role {
    name := self.roleName;
}
```

Figure 26: Excerpt of a model-to-model transformation definition (figure taken from [Berger, 2016]).

Consolidation of the different parts into a unified high-level instance model

Until now, we have three different parts of the high-level instance model, the transformed low-level instance model, the workflow part (including the mask instances) and the authorization constraints, which have been defined in the high-level instance model (e.g., concrete instances of static SoD, cardinality and prerequisite role constraints).

In this step, these parts are integrated into a uniform high-level instance model. The linking process is accomplished via the type and name of the elements/instances. For example, authorization constraint instances refer to concrete role instances, e.g., an SoD constraint may refer to the roles `umb_chief` and `umb_admin`. Consequently, the extracted roles `umb_chief` and `umb_admin` must be unified with the corresponding counterparts in the aforementioned SoD constraint.

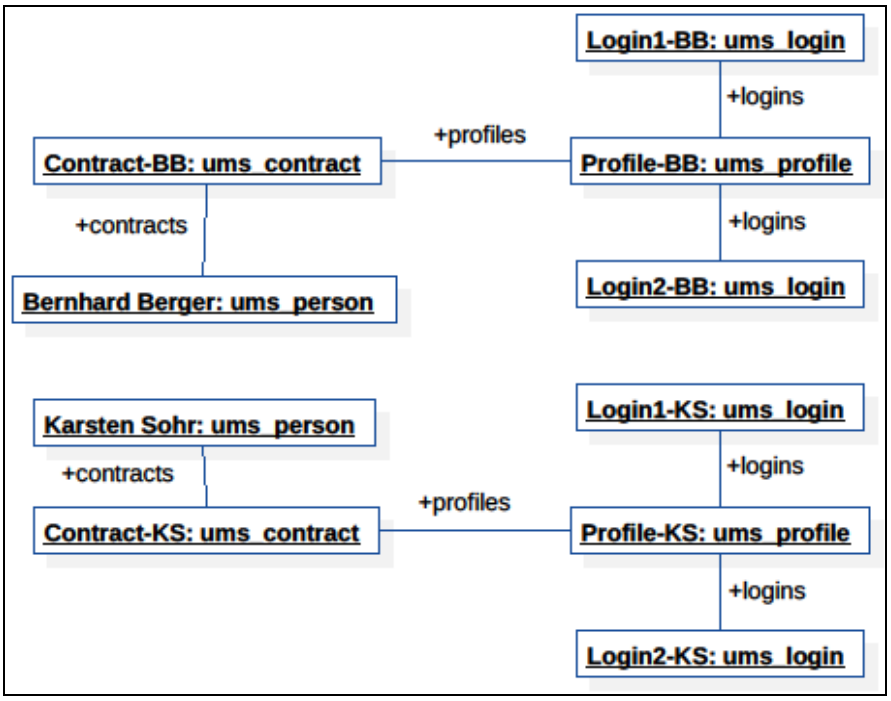

Figure 27: An example low-level instance model (figure taken from [Berger, 2016]).

Figure 28: The instance model from Figure 27 after the transformation (figure taken from [Berger, 2016]).

Data reduction

The consolidated high-level instance model shall be analyzed with the UML validation tool [Przigoda et al., 2015]. As the validation tool consumes much storage, we first simplified the current model to make this step feasible. In particular, we had to remove information that was not needed for validation, such as name attributes for all instances. Again, we used QVTo to transform the original model into the reduced version. In addition, we defined a mapping that connected the original with the reduced model. This mapping was needed for generating meaningful error messages (aspect of usability).

Validation of the consolidated high-level instance model

In the final step, the consolidated instance model can be tested with a UML validation tool. The principles of this task have already been described in Section 4.2.3 and hence we do not discuss this point in detail here. Instead of the USE tool, however, we employed the DFKI tool [Przigoda et al., 2015] for validation because it directly supports Ecore models. The DFKI tool is based on Satisfiability Modulo Theories (SMT) and translates SMT problems to a SAT prover [Soeken et al., 2010].

As already mentioned in Section 4.2.3, typical validation problems that can be addressed with our approach include checking whether the consolidated high-level model violates OCL constraints (authorization constraints). For example, a user being assigned to both the roles `umb_chief` and `umb_admin` would violate the aforementioned static SoD constraint.

We also carry out a simplified reflexion analysis [Murphy et al, 1995] within the validation step. This analysis attempts to identify dependences between workflow steps, which appear in the implementation (mask dependences), but do not occur in the business process descriptions and vice versa.

5.3 Experience with the Approach and Open Problems

We have realized the analysis pipeline mentioned before with the help of several small demonstrators to show the principle feasibility of our approach. Several steps of this methodology can be reused in similar projects such that the approach can be generalized. For example, the extraction of the domain data model can be reused in other projects because we use common JPA interfaces for regaining this information from Java source/binary code. Also, the extraction of the workflow information is largely reusable as the static analysis is based upon the Seam and JSF frameworks, which are employed in the implementation of the B2BApplication, but are also widely used. Furthermore, the consolidation, reduction and validation steps are mostly independent of the specific application under analysis.

Although the main approach worked and it was possible to extract and validate role-based policies from a real-world business application, several points remain open:

1. We extracted concrete role and permission names from the source code of the analyzed application. It is also possible to directly access the database to regain this information. This application database contains roles, permissions and users. Direct access to the database allows us to directly obtain the corresponding low-level information. However, we cannot currently carry out this step because we do not have access to the productive system.
2. Some steps carried out to analyze the B2BApplication are application-specific and therefore cannot be directly applied to other applications. In particular, we utilized specific authorization API calls, which are used within the B2BApplication, such as `hasPermission()` method calls. It would have been better if the application had used common authorization APIs, e.g., `EJBContext.isCallerInRole()`, which is made available by JEE. Then the approach could have been applied to each software that uses these authorization APIs. The given application, however, has not been developed with having in mind the ability to be analyzable.
3. To perform a comprehensive analysis of the application, all used authorization checks must be identified. In addition, it must be clarified whether these checks are actually used to enforce a role-based policy. For example, in the current implementation some checks are employed to merely control the visibility of GUI elements rather than restricting access to data or process steps. In particular, it is currently unclear whether additional access control checks exist that further restrict data access.
4. Some advanced access control concepts, such as context constraints [Georgiadis et al., 2001], are not supported by our analyses, although the business application is supposed to use them. The fact that not all authorization concepts are supported makes the analysis results somewhat misleading.
5. The static analyses are incomplete in the sense that not all software framework concepts are currently supported. In particular, this observation applies to the workflow extraction step, in which the Seam framework is not fully implemented.
6. The DFKI OCL validation tool [Przigoda et al., 2015] does not cover the complete OCL standard, which prevented us from performing a comprehensive validation. It would be important to extend the DFKI tool to support OCL more completely. Also, a translation of the Ecore models into UML models would be useful—then the USE tool could be applied in the validation step, which would allow us to compare the functionality and efficiency of both tools.

The main lesson learned from this case study is the observation that the approach seems to be most powerful if the application under analysis has been developed with analysis in mind. In this respect, using standard security APIs would be helpful. Also, authorization checks should be called at the beginning of a protected method rather than somewhere deep in the call hierarchy. This would simplify the analysis step considerably.

5.4 Summary of this Work

The approach introduced in this chapter combines the architectural risk analysis based on reverse-engineering techniques and the definition of a systematic process to managing role-based policies in security-critical software. In particular, we applied our analysis techniques to a relevant and security-critical business application of a large enterprise.

Our experience with the approach shows that it is important to make available a methodology to analyze and manage role-based policies in legacy code including adequate tool support. Our proposed analysis methodology, albeit still being at the demonstrator level, shows promise to be generalizable. This is even more relevant as many organizations, e.g., large enterprises, government agencies, and financial institutes, run security-critical proprietary applications, where the implemented security architecture is often undocumented. Here, our proposed approach can help one better understand and finally improve the security status of the whole application landscape of an organization.

The contribution of this habilitation thesis work lies in defining the general research agenda for managing and analyzing role-based policies in an application landscape (combining the specification and validation of role-based policies and the reverse engineering approach). In addition, the underlying access control concepts have been elaborated, which formed the basis for the metamodel introduced in Section 5.1.

Beyond the technical description [Berger, 2016], a recent publication summarizes the approach introduced in this chapter [Berger et al., 2019]. This paper, which is also part of this habilitation thesis, additionally discusses the modeling and reverse engineering approach in the context of a healthcare application and a port community system (the latter case study is related to the BMBF-funded research project PortSec).

Chapter 6

Summary and Outlook

In this thesis, we discussed different aspects of architectural risk analysis in detail. We highlighted the relevance of this topic, although currently industry often seems to neglect architectural aspects within their software development processes. In particular, we presented different approaches to supporting security architects in developing and analyzing security architectures for their software systems.

With the help of our reverse engineering-based approach, we automatically extracted the security architecture from Java source or binary code and represented it in form of dataflow diagrams (graphs) as well as DBC specifications. The former representation allows us to visualize the security architecture such that an analyst can better comprehend it and identify possible weaknesses. The formal representation as DBC specifications allows an analyst to validate the specifications and check the source code against candidate specifications. This may improve the quality of the implemented security architecture.

We also proposed a forward-engineering approach to architectural risk analysis, with a focus on the specification and analysis of role-based policies. In particular, we demonstrated how to specify such policies in UML/OCL without confronting the user (usually, a security officer) with OCL constraints. To achieve this goal, we used concepts of UML metamodeling. We currently support the main RBAC concepts as well as advanced aspects, such as history-based SoD constraints and role-based delegation mechanisms. Context constraints, however, are still missing due to uncertainty to adequately generalize this concept.

Moreover, we combined both topics by automatically extracting the implemented role-based policy of a real-world business application with the help of static program-analysis techniques. The extracted policy can then be analyzed and consolidated. Thereafter, the involved enterprise can better understand and maybe change their role-based policies, which are implemented in their applications. As this topic was pursued within the frameworks of a feasibility study with this large enterprise, we had the chance to evaluate our concepts in practice. Even more, this task let us identify open research topics in this area. One open research topic is to build a real

prototype, which security officers can evaluate with respect to effectiveness and usability. Another is how to build software that is better amenable to our analysis techniques.

Similar remarks hold for the tool environment to extract DFDs from Java-based applications. This analysis environment must be further developed to cover the currently supported software frameworks Android and JEE more completely. Also, other software frameworks, such as Spring and Struts, and even other programming languages, including C/C++ and C#, should be supported to provide an analysis platform that is powerful enough to be of use for real-world projects. Furthermore, the graphical user interface and usability in general must be significantly improved. Then it would be possible to conduct comprehensive and interesting experiments, which hopefully demonstrate the efficiency and effectiveness of our approach to architectural risk analysis. This important step, however, can only be carried out if substantial funding for a validation project is available.

At any rate, this habilitation thesis lays out the foundation for more research in the field of architectural risk analysis of software, which is an increasingly important aspect of an effective SDL. We truly believe that our work will lead—if appropriately extended—to more systematic approaches to architectural risk analysis and in the end to more secure software.

Bibliography

[Ahn, 1999] G.-J. Ahn. The RCL 2000 language for specifying role-based authorization constraints. Doctoral Thesis, George Mason University, Virginia, 1999.

[American National Standards Institute, 2004] American National Standards Institute Inc. Role Based Access Control. American National Standard for Information Technology, ANSI-INCITS 359-2004, 2004.

[Anderson, 2008] R. Anderson. Security engineering (2nd edition). John Wiley & Sons, 2008.

[Appel and Palsberg, 2003] A. W. Appel, J. Palsberg. 2003. Modern Compiler Implementation in Java (2nd edition). Cambridge University Press, New York, NY, USA.

[Arce et al., 2014] I. Arce, K. Clark-Fisher, N. Daswani, J. DelGrosso, D. Dhillon, C. Kern, T. Kohno, C. Landwehr, G. McGraw, B. Schoenfield, M. Seltzer, D. Spinellis, I. Tarandach, J. West. Avoiding the top 10 software security design flaws. IEEE Computer Society, 2014. Accessible under: https://www.computer.org/cms/CYBSI/docs/Top-10-Flaws.pdf

[Backes et al., 2016] M. Backes, S. Bugiel, E. Derr, P. McDaniel, D. Octeau, S.Weisgerber. On demystifying the android application framework: re-visiting android permission specification analysis. In: 25th USENIX Security Symposium (USENIX Security 2016), pp. 1101–1118. USENIX Association, Austin (2016)

[Barka and Sandhu, 2000] E. Barka, R.S. Sandhu. Framework for role-based delegation models. In Proceedings of the 16th Annual Computer Security Applications Conference, December 2000.

[Bartel et al., 2014] A. Bartel, J. Klein, M. Monperrus and Y. Le Traon: Static Analysis for Extracting Permission Checks of a Large Scale Framework: The Challenges And Solutions for Analyzing Android. In IEEE Transactions of Software Engineering, 2014.

[Bellovin, 2016] S. M. Bellovin. Thinking Security: Stopping Next Year's Hackers. Addison-Wesley, Boston, 2016.

[Berger, 2016] B. Berger. Ergebnisbericht. DFKI GmbH, August 2016.

[Berger et al., 2013] B. Berger, K. Sohr, R. Koschke. Extracting and Analyzing the Implemented Security Architecture of Business Applications. In Proc. of the 17th European Conference on Software Maintenance and Reengineering (CSMR 2013), Genua, Italy, 2013.

[Berger et al., 2014] B. Berger, K. Sohr, U. H. Kalinna. Architekturelle Sicherheitsanalyse für Android Apps. In D-A-CH Security 2014. Bestandsaufnahme - Konzepte - Anwendungen - Perspektiven. Seiten 287 - 297. syssec. ISBN 978-3-00-046463-8. 2014.

[Berger et al., 2016] B. Berger, K. Sohr, R. Koschke. Automatically Extracting Threats from Extended Data Flow Diagrams. In Proc. 8th International Symposium on Engineering Secure Software and Systems (ESSoS 2016), London, April 2016.

[Berger et al., 2019] B. Berger, C. Maeder, R. Wete Nguempnang, K. Sohr, C. Rubio-Medrano. Towards Effective Verification of Multi-Model Access Control Properties. In Proc. of the 2019 ACM Symposium on Access Control Models and Technologies (SACMAT), Toronto, Canada, June 2019.

[Bunke and Sohr, 2020]. M. Bunke, K. Sohr. Towards Supporting Software Assurance Assessments by Detecting Security Patterns. Software Quality Journal, Springer, 2019. To appear.

[Burdy et al., 2005] L. Burdy, Y. Cheon, D.R. Cok, M.D. Ernst, J.R. Kiniry, G.T. Leavens, K.R.M. Leino, E. Poll. An overview of JML tools and applications. International Journal on Software Tools for Technology Transfer 7(3), 212–232 (2005).

[Chess, 2002] B. Chess. 2002. Improving Computer Security Using Extended Static Checking. In Proceedings of the 2002 IEEE Symposium on Security and Privacy (SP '02). IEEE Computer Society, Washington, DC, USA, 160-.

[Chess and West, 2007] B. Chess, J. West. Secure Programming with Static Analysis (First edition). Addison-Wesley Professional (2007).

[Chin et al., 2011] E. Chin, A. Porter Felt, K. Greenwood, D. Wagner. 2011. Analyzing inter-application communication in Android. In Proceedings of the 9th International Conference on Mobile systems, applications, and services (MobiSys '11). ACM, New York, NY, USA, 239-252.

[Cok, 2011] D. Cok. OpenJML: JML for Java 7 by Extending OpenJDK. In NASA Formal Methods, pages 472-479. Volume 6617 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, 2011.

[Cok, 2017] D. Cok: Response—Problems with String and ESC #544, 2017. Accessible under: https://github.com/OpenJML/OpenJML/issues/544

[Common Criteria, 2009] Common Criteria: Common Criteria for Information Technology Security Evaluation—Part 3: Security assurance components (2009), accessible under: http://www.commoncriteriaportal.org/files/ccfiles/CCPART3V3.1R3.pdf

[Cyl, 2019] M. Cyl. Sicherheitsanalyse für Android-Systemdienste auf der Basis von Program Slicing, Master thesis, University of Bremen, 2019.

[Danezis et al., 2015] G. Danezis, J. Domingo-Ferrer, Marit Hansen, Jaap-Henk Hoepman, Daniel Le Métayer, R. Tirtea,, S. Schiffner. Privacy and data protection by design - from policy to engineering. ENISA Technical Report, January 2015.

[Detmers, 2016] M. Detmers. Evaluation des WALA-Slicers bzgl. der Anwendbarkeit auf sicherheits-kritische Java-Programme. University of Bremen, Germany, 2016.

[Dhillon, 2011] D. Dhillon. Developer-driven threat modeling: Lessons learned in the trenches. IEEE Security & Privacy 4 (2011): 41-47.

[Dierks and Rescorla, 2008] T. Dierks, E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. Request for Comments 5246, 2008. Accessible under: https://www.ietf.org/rfc/rfc5246.txt

[DLR, 2010] Deutsches Zentrum für Luft- und Raumfahrt e. V. (DLR). ASKS - Architekturbasierte Sicherheitsanalyse geschäftskritischer Software-Systeme, 2010. Accessible under: http://www.softwaresysteme.pt-dlr.de/media/content/Infoblatt_ASKS.pdf

[Dolby and Sridharan, 2010] J. Dolby, M. Sridharan: Static and Dynamic Program Analysis Using WALA, 2010. Accessible under: http://wala.sourceforge.net/files/PLDI_WALA_Tutorial.pdf

[Eckert, 2013] C. Eckert. IT-Sicherheit: Konzepte-Verfahren-Protokolle. Walter de Gruyter, 2013.

[Egele et al., 2013] M. Egele, D. Brumley, Y. Fratantonio, C. Kruegel. An empirical study of cryptographic misuse in android applications. In Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security (CCS '13). ACM, New York, NY, USA, 73-84.

[Enck et al., 2009] W. Enck, M. Ongtang, P. McDaniel. Understanding Android Security. IEEE Security and Privacy 7, 1 (January 2009)

[Ernst et al., 2007] M.D. Ernst, J.H. Perkins, P.J. Guo, S. McCamant, C. Pacheco, M.S. Tschantz, C. Xiao: The Daikon system for dynamic detection of likely invariants. Sci. Comput. Program. 69, 35–45, December 2007.

[Fahl et al., 2012] S. Fahl, Harbach, T. Muders, M. Smith, L. Baumgärtner, B. Freisleben: Why Eve and Mallory love Android: An analysis of Android SSL (in)security. In: Proceedings of the 2012 ACM Conf. on Computer and communications security. pp. 50–61., New York, USA (2012).

[Ferraiolo et al., 2007] D. F. Ferraiolo, D. R. Kuhn, R. Chandramouli. 2007. Role-Based Access Control (2nd edition). Artech House, Inc., Norwood, MA, USA.

[Flanagan et al., 2002] C. Flanagan, K.R.M. Leino, M. Lillibridge, G. Nelson, J.B. Saxe, R. Stata. Extended static checking for Java. In: Proc. of the ACM Conference on Programming language design and implementation. pp. 234–245. PLDI ’02 (2002).

[Georgiadis et al., 2001] C.K. Georgiadis, I. Mavridis, G. Pangalos, R.K. Thomas. Flexible team-based access control using contexts. In Proceedings of the ACM Symposium on Access Control Models and Technologies, pp. 21-27, May 2001.

[Georgiev et al., 2012] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, V. Shmatikov. 2012. The most dangerous code in the world: validating SSL certificates in non-browser software. In Proceedings of the 2012 ACM conference on Computer and communications security (CCS '12). ACM, New York, NY, USA, 38-49.

[Gerken, 2015] P. Gerken. Statische Sicherheitsanalyse von Java Enterprise-Anwendungen mittels Program Slicing, Bachelor thesis, University of Bremen, 2015.

[Glander, 2017] T. Glander. Personal communication, 2017

[Gligor et al., 1998] V.D. Gligor, S.I. Gavrila, D. Ferraiolo. On the formal definition of separation-of-duty policies and their composition. In Proceedings of the 1998 IEEE Symposium on Security and Privacy, pp. 172-185, 1998.

[Gogolla et al., 2007] M. Gogolla, F. Büttner, M. Richters. 2007. USE: A UML-based specification environment for validating UML and OCL. Sci. Comput. Program. 69, 1-3 (December 2007), 27-34.

[Gogolla et al., 2014] M. Gogolla, L. Hamann, F. Hilken, M. Kuhlmann, R. B. France. From Application Models to Filmstrip Models: An Approach to Automatic Validation of Model Dynamics. In Hans-Georg Fill, Dimitris Karagiannis, and Ulrich Reimer, editors, Proc. Modellierung (MODELLIERUNG' 2014), pages 273-288. GI, LNI 225, 2014.

[Gorski et al., 2019] S. A. Gorski, B. Andow, A. Nadkarni, S. Manandhar, W. Enck, E. Bodden, A. Bartel. 2019. ACMiner: Extraction and Analysis of Authorization Checks in Android's Middleware. In Proceedings of the Ninth ACM Conference on Data and Application Security and Privacy (CODASPY '19). ACM, New York, NY, USA, 25-36.

[Green and Smith, 2016] M. Green, M. Smith. Developers are Not the Enemy!: The Need for Usable Security APIs, IEEE Security & Privacy vol. 14 no. 5, p. 40-46, Sept.-Oct., 2016

[Gulmann, 2014] M. Gulmann. Statische Sicherheitsanalyse der Android Systemservices. Master thesis, University of Bremen, Germany, 2014. Accessible under:

http://www.informatik.uni-bremen.de/~sohr/papers/Gulmann.pdf

[Hernan et al., 2006] S. Hernan, S. Lambert, T. Ostwald, A. Shostack. Uncover Security Design Flaws Using the STRIDE Approach. MSDN Magazine, Nov. 2006.

[Hewlett-Packard, 2016]. Hewlett-Packard Development Company. Fortify Static code analyzer, 2016. Accessible under: http://www8.hp.com/de/de/software-solutions/static-code-analysis-sast/index.html

[Hatcliff et al., 2012] J. Hatcliff, G.T. Leavens, K. R. M. Leino, P. Müller, M. Parkinson. 2012. Behavioral interface specification languages. ACM Comput. Surv. 44, 3, Article 16 (June 2012).

[Hilken et al., 2014] F. Hilken, L. Hamann, M. Gogolla. Transformation of UML and OCL models into filmstrip models. International Conference on Theory and Practice of Model Transformations. Springer International Publishing, 2014.

[Hillmann, 2015] K.T. Hillmann. Sicherheitsanalyse von App-gesteuerten Alarmanlagen, Master thesis, University of Bremen, September 2015.

[Horwitz et al., 1990] S. Horwitz, T. Reps, D. Binkley. 1990. Interprocedural slicing using dependence graphs. ACM Trans. Program. Lang. Syst. 12, 1 (January 1990), 26-60.

[Howard and Lipner, 2006] Michael Howard and Steve Lipner. 2006. The Security Development Lifecycle. Microsoft Press, Redmond, WA, USA.

[IBM, 2016] IBM Corp. IBM Security AppScan Source, 2016. Accessible under: http://www-03.ibm.com/software/products/de/appscan-source

[ICS-CERT, 2015] ICS-CERT. Advisory (ICSA-15-064-05) Siemens SPCanywhere App Vulnerabilities, 2015. Accessible under: https://ics-cert.us-cert.gov/advisories/ICSA-15-064-05

[International Organization for Standardization, 2011] International Organization for Standardization. ISO/IEC 27034:2011+ Information technology — Security techniques — Application security, 2011. Accessible under: http://www.iso27001security.com/html/27034.html

[Joshi et al., 2005] J. B. D. Joshi, E. Bertino, U. Latif, A. Ghafoor. 2005. A Generalized Temporal Role-Based Access Control Model. IEEE Trans. on Knowl. and Data Eng. 17, 1 (January 2005), 4-23.

[Koscher et al., 2010] K. Koscher, A. Czeskis, F. Roesner, S. Patel, T. Kohno, S. Checkoway, S. Savage. Experimental security analysis of a modern automobile. In Security and Privacy (SP), 2010 IEEE Symposium on (pp. 447-462). IEEE (2010).

[Krinke, 2003] J. Krinke. Advanced Slicing of Sequential and Concurrent Programs, Universität Passau, Germany, April 2003.

[Langner, 2013] R. Langner. To Kill a Centrifuge, A Technical Analysis of – What Stuxnet's Creators Tried to Achieve, November 2013. Accessible under: http://www.langner.com/en/wp-content/uploads/2013/11/To-kill-a-centrifuge.pdf

[Leavens et al., 2006] G.T. Leavens, A.L. Baker, C. Ruby. Preliminary design of JML: a behavioral interface specification language for Java. SIGSOFT Softw. Eng. Notes 31, 3 (Mai 2006), 1-38.

[Liang et al., 2016] T. Liang, A. Reynolds, N. Tsiskaridze, C. Tinelli, C. Barrett, Mo. Deters. 2016. An efficient SMT solver for string constraints. Form. Methods Syst. Des. 48, 3 (June 2016), 206-234.

[Liebig, 2014] C. Liebig: Sicherheitsanalyse von mobilen Geschäftsanwendungen, Masterarbeit, University of Bremen, 2014. Accessible under: http://www.informatik.uni-bremen.de/~sohr/papers/MasterthesisLiebig.pdf

[Meyer, 1989] B. Meyer. From structured programming to object-oriented design: The road to Eiffel. Structured Programming (1), 19–39 (1989).

[McGraw, 2006] G. McGraw. Software Security. Building Security In. Addison-Wesley, 2006.

[McGraw, 2013] G. McGraw. Software [in]security and scaling architecture risk analysis. TechTarget, December 2013. Accessible under: http://searchsecurity.techtarget.com/opinion/McGraw-Software-insecurity-and-scaling-architecture-risk-analysis

[McGraw, 2017] G. McGraw. 2017. Six Tech Trends Impacting Software Security. IEEE Computer 50, 5 (May 2017), 100-102.

[McGrew and Viega, 2004] D. McGrew, J. Viega. The Security and Performance of the Galois/Counter Mode (GCM) of Operation. Proceedings of INDOCRYPT ’04, Springer-Verlag, 2004.

[Microsoft, 2010] Microsoft Corp. Security Development Lifecycle – Simplified Implementation of the Microsoft SDL, 2010. Accessible under: https://www.microsoft.com/en-us/download/details.aspx?id=12379

[MITRE, 2014] MITRE Corporation. The Common Weakness Enumeration (CWE) Initiative, 2014. Accessible under: http://cwe.mitre.org/

[MITRE, 2015] MITRE Corporation. Common Attack Pattern Enumeration and Classification (CAPEC), 2015. Accessible under: http://capec.mitre.org/

[Müller, 2015] D. Müller. Untersuchung zur Broadcast-Sicherheit in Android-Apps, Bachelor Thesis, University of Bremen, Germany, 2015. Accessible under: http://www.informatik.uni-bremen.de/~sohr/papers/BachelorThesisDaniel.pdf

[Murphy et al., 1995] G. C. Murphy, D. Notkin, K. Sullivan. 1995. Software reflexion models: bridging the gap between source and high-level models. In Proceedings of the 3rd ACM SIGSOFT symposium on Foundations of software engineering (SIGSOFT '95), Gail E. Kaiser (Ed.). ACM, New York, NY, USA, 18-28.

[Mustafa and Sohr, 2015] T. Mustafa, K. Sohr. 2015. Understanding the implemented access control policy of Android system services with slicing and extended static checking. Int. J. Inf. Secur. 14, 4 (August 2015), 347-366.

[Nash and Poland, 1990] M.J. Nash, K.R. Poland. Some conundrums concerning separation of duty. In Proceedings of the 10th IEEE Symposium on Research in Security and Privacy, 201-207, 1990.

[Object Management Group, 2007] Object Management Group Inc. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, 2007. Accessible under: http://www.omg.org/cgi-bin/doc?ptc/07-07-07.pdf

[O'Connor and Loomis, 2010] A.C. O'Connor, R.J. Loomis. Economic analysis of role-based access control. Final Report, National Institute of Standards and Technology, 2010.

[Organization for the Advancement of Structured Information Standards, 2013] Organization for the Advancement of Structured Information Standards. eXtensible Access Control Markup Language (XACML) Version 3.0, 2013.
Accssible under: http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-cs-01-en.pdf.

[Porter Felt et al., 2011] A. Porter Felt, E. Chin, S. Hanna, D. Song, D. Wagner. Android permissions demystified. In: Proc. of the 18th ACM Conference on Computer and Communications Security, CCS 2011, Chicago, I, USA. pp. 627–638. ACM (2011).

[PortSec, 2016] PortSec-Konsortium. PortSec-2: IT-Risikomanagement in der Hafentelematik auf Basis der Software-Architektur, BMBF proposal, Bremen, 2016

[Przigoda et al., 2015] N. Przigoda, R. Wille, R. Drechsler. 2015. Contradiction Analysis for Inconsistent Formal Models. In Proceedings of the 2015 IEEE 18th International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS '15). IEEE Computer Society, Washington, DC, USA, 171-176.

[Quante and Koschke, 2008] J. Quante, R. Koschke. 2008. Dynamic object process graphs. J. Syst. Softw. 81, 4 (April 2008), 481-501.

[Saltzer and Schroeder, 1975] J.H. Saltzer, M.D. Schroeder. The protection of information in computer systems. Proceedings of the IEEE, 63(9), 1278-1308, 1975.

[Sandhu, 1996] R. S. Sandhu. Role Hierarchies and Constraints for Lattice-Based Access Controls. In Proceedings of the 4th European Symposium on Research in Computer Security: Computer Security (ESORICS '96). Springer, London, UK, UK, 65-79, 1996.

[Sandhu et al., 1996] R.S. Sandhu, E.J. Coyle, H.L. Feinstein, C.E. Youman. Role-based access control models, IEEE Computer 29(2), 38-47, 1996.

[SAP, 2019] SAP SE. The Secure Software Development Lifecycle at SAP, 2019. Accessible under: https://www.sap.com/documents/2016/03/a248a699-627c-0010-82c7-eda71af511fa.html

[SAP, 2015] SAP Product Security Response. Multiple vulnerabilities in SAP Mobile Document Android Client. Security Note 2194730, October 2015, Accessible under: https://www.onapsis.com/blog/analyzing-sap-security-notes-october-2015

[Schaad et al., 2006] A. Schaad, V. Lotz, K. Sohr. A Model-checking Approach to Analysing Organisational Controls in a Loan Origination Process. In 11th ACM Symposium on Access Control Models and Technologies, Lake Tahoe, CA, 2006.

[Shao et al., 2016] Y. Shao, Q. A. Chen, Z. M. Mao, J. Ott, Z. Qian. Kratos: discovering inconsistent security policy enforcement in the android framework. In: Proceedings of the 23rd NDSS The Network and Distributed System Security Symposium, San Diego (2016)

[Shostack, 2014] A. Shostack. Threat modeling: Designing for security. John Wiley & Sons, 2014.

[Siemens ProductCERT, 2015] Siemens ProductCERT. SSA-31141: Incorrect Certificate Verification in Android App HomeControl for Room Automation, 2015. Accessible under: http://www.siemens.com/innovation/pool/de/forschungsfelder/siemens_security_advisory_ssa-311412.pdf

[Siemens ProductCERT, 2018] SSA-468514: Improper Certificate Validation Vulnerability in VMS Video Mobile App for Android and iOS, 2018. Accessible under: https://cert-portal.siemens.com/productcert/pdf/ssa-468514.pdf

[Simon and Zurko, 1997] R. Simon, M. Zurko. Separation of duty in role-based environments. In Proceedings of the 10th IEEE Computer Security Foundations Workshop (CSFW'97), 183-194, June 1997.

[Sindre and Opdahl, 2005] G. Sindre, A. L. Opdahl. 2005. Eliciting security requirements with misuse cases. Requirements Engineering 10, 1 (January 2005), 34-44.

[Soeken et al., 2010] M. Soeken, R. Wille, M. Kuhlmann, M. Gogolla, R. Drechsler. 2010. Verifying UML/OCL models using Boolean satisfiability. In Proceedings of the Conference on Design, Automation and Test in Europe (DATE '10). European Design and Automation Association, 3001 Leuven, Belgium, 1341-1344.

[Sohr and Berger, 2010] K. Sohr, B. Berger. Idea: Towards Architecture-Centric Security Analysis of Software. Proc. 2nd International Symposium on Engineering Secure Software and Systems (ESSoS 2010), Pisa, Italy.

[Sohr and Maeder, 2019] K. Sohr, C. Maeder. KMU-innovativ - Verbundprojekt: IT-Risikomanagement in der Hafentelematik auf Basis der Software-Architektur – PortSec-2 – Teilvorhaben: Automatische Erstellung und Sicherheitsanalyse der Systemarchitektur für die Hafentelematik, Schlussbericht der Universität Bremen, February 2019.

[Sohr and Schröer, 2019] K. Sohr, M. Schröer. KMU-innovativ - Verbundprojekt: Entwicklung sicherer mobiler Anwendungen zur Steuerung von Smarthome-Systemen – SecureSmartHomeApp – Teilvorhaben: Dynamische und statische Sicherheitsanalysen von App-gesteuerten Smarthome-Systemen, Schlussbericht der Universität Bremen, June 2019.

[Sohr et al., 2005] K. Sohr, M. Drouineaud, G.-J. Ahn. Formal specification of role-based security policies for clinical information systems. In Proceedings of the 20th ACM Symposium on Applied Computing, Santa Fe, New Mexico, 2005.

[Sohr et al., 2008] K. Sohr, M. Drouineaud, G.-J. Ahn, M. Gogolla. Analyzing and Managing Role-Based Access Control Policies. IEEE Transactions on Knowledge and Data Engineering, Vol. 20, No. 7, 2008.

[Sohr et al., 2008a] K. Sohr, T. Mustafa, G.-J. Ahn, X. Bao. Enforcing Role-Based Access Control Policies in Web Services with UML and OCL, 24th Annual Computer Security Applications Conference, Anaheim CA, December 2008.

[Sohr et al., 2011] K. Sohr, T. Mustafa, A. Nowak. Software Security Aspects of Java-Based Mobile Phones. In Proceedings of the 26th ACM Symposium on Applied Computing, Taichung, Taiwan, 2011.

[Sohr et al., 2012] K. Sohr, M. Kuhlmann, M. Gogolla, H. Hu, G.-J.Ahn. Comprehensive Two-Level Analysis of Role-Based Delegation and Revocation Policies with UML and OCL. In Information and Software Technology (IST), Volume 54, No. 12, 2012.

[Sohr et al., 2016] K. Sohr, T. Mustafa, M. Gulmann, P. Hirch. Towards Security Program Comprehension with Design by Contract and Slicing, Technical Report No. 80, Center for Computing and Communications Technologies (TZI), 2016. Accessible under: http://www.tzi.de/fileadmin/resources/publikationen/tzi_berichte/TZI-Report_80.pdf

[Sridharan, 2017] M. Sridharan. Slicer is missing statements for random number example. Issue #165, 2017. Accessible under: https://github.com/wala/WALA/issues/165

[Steinberg et al., 2009] D. Steinberg, F. Budinsky, M. Paternostro, E. Merks. 2009. Emf: Eclipse Modeling Framework 2.0 (2nd edition). Addison-Wesley Professional.

[Synopsis, 2016] Synopsys Inc. Coverity® - Static Code Analysis, 2016. Accessible under: https://www.synopsys.com/software/coverity/Pages/default.aspx#

[Vallée-Rai et al., 1999] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, Sundaresan. Soot—a Java bytecode optimization framework. In Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research (CASCON '99), Stephen A. MacKay and J. Howard Johnson (Eds.). IBM Press, 1999.

[Vanhoef and Piessens, 2017] M. Vanhoef, F. Piessens. Key Reinstallation Attacks: Forcing Nonce Reuse in WPA2. In Proceedings of the 24th ACM Conference on Computer and Communication Security (CCS 2017), Dallas, USA, 30 October - 3 November 2017.

[VDIVDE-IT, 2014] VDI/VDE Innovation + Technik GmbH. Zertifizierte Sicherheit für mobile Anwendungen (ZertApps), 2016. Accessible under: http://www.vdivde-it.de/KIS/kmu-innovativ/zertapps

[VDIVDE-IT, 2016] VDI/VDE Innovation + Technik GmbH. IT-Risikomanagement in der Hafentelematik auf Basis der Software-Architektur (PortSec). Accessible under: http://www.vdivde-it.de/KIS/kmu-innovativ/portsec

[Wainer and Kumar, 2005] J. Wainer, A. Kumar. 2005. A fine-grained, controllable, user-to-user delegation method in RBAC. In Proceedings of the 10th ACM symposium on Access control models and technologies (SACMAT '05). ACM, New York, NY, USA, 59-66.

[Warmer and Kleppe, 2013] J. Warmer, A. Kleppe. 1998. The Object Constraint Language: Precise Modeling with UML. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[Weiser, 1981] M. Weiser. Program slicing. In: Proceedings of the International Conference on Software Engineering. pp. 439–449. IEEE Press, Piscataway, NJ, USA (1981)

[Zhang et al., 2003] L. Zhang, G.-J. Ahn, B.-T. Chu. A rule-based framework for role-based delegation and revocation. ACM Transactions on Information and Systems Security and Privacy, 6(3), 2003.

[Ziegler, 2015] H. Ziegler. Analyse der Verwendung von Kryptographie-APIs in Java-basierten Anwendungen. Master thesis, University of Bremen, Germany, 2015.

[Ziemann and Gogolla, 2003] P. Ziemann, M. Gogolla. OCL extended with temporal logic. International Andrei Ershov Memorial Conference on Perspectives of System Informatics. Springer Berlin Heidelberg, 2003.

Appendix A

Lectures

1. Spring 2005: Information security I (with Prof. Dr. Carsten Bormann)
2. Spring 2007: Information security I (with Prof. Dr. Carsten Bormann)
3. Fall 2008: Information security I (with Prof. Dr. Carsten Bormann)
4. Fall 2009: Information security I (with Prof. Dr. Carsten Bormann)
5. Fall 2010: Information security I (with Prof. Dr. Carsten Bormann)
6. Fall 2011: Information security I (with Prof. Dr. Carsten Bormann)
7. Fall 2012: Information security I (with Prof. Dr. Carsten Bormann)
8. Fall 2013: Information security I (with Prof. Dr. Carsten Bormann)
9. Fall 2014: Information security I (with Prof. Dr. Carsten Bormann)
10. Fall 2015: Information security I (with Prof. Dr. Carsten Bormann and Prof. Dr. Tim Güneysu)
11. Fall 2016: Information security I (with Prof. Dr. Carsten Bormann and Prof. Dr. Tim Güneysu)
12. Fall 2017: Information security I (with Prof. Dr. Carsten Bormann)
13. Fall 2018: Information security I (with Prof. Dr. Carsten Bormann)
14. Fall 2018: How does IT work? (with Prof. Dr. Thomas Kemmerich, Dr. Dennis-Kenji Kipker)
15. Fall 2019: Information security I (with Prof. Dr. Carsten Bormann)
16. Fall 2019: Usable security (seminar with Prof. Dr. Rainer Malaka, Mehrdad Bahrini)

Appendix B

Successfully Supervised Doctoral Theses

1. Dr. Tanveer Mustafa. Static Security Analysis of Java Applications with an Approach Based on Design by Contract, University of Bremen, Germany, 2013 (also Committee Member)
2. Dr. Michaela Bunke. Security-Pattern Recognition and Validation, University of Bremen, Germany, 2019 (also Second Reviewer)

Appendix C

Supervised Bachelor, Master and Diploma Theses as First Reviewer

Bachelor

1. Kai Hillmann. Darstellung und Analyse eines Konzeptes zur digitalen Beweissicherung, 2011
2. Philipp Nguyen. NFC-Sicherheit mit Smartphones – Sicherheitsanalyse von Android-Applikationen mit NFC-Funktionalität, 2013
3. Markus Gulmann. Sicherheitsanalyse ausgewählter Systemservices des mobilen Betriebssystems Android, 2013
4. Alexander Neer. Richtlinien für den sicheren SSL/TLS-Einsatz, 2013
5. Malte Kuhn. Anomalieerkennung von Applikationsverhalten auf Android, 2014
6. Malte Batram. Dynamische Sicherheitsanalyse von ActionScript-basierten Webanwendungen, 2014
7. Denis Szadkowski. Evaluation eines Werkzeugs zur statischen Analyse von SSL/TLS-Schwachstellen, 2014
8. Patrick Hofmann. Entwicklung einer modularen Pentetration-Test-Suite zur Sicherheitsanalyse auf Android-Geräten, 2014
9. Alex Antoni. Darstellung von Ergebnissen statischer Codeanalysen installierter Android apk-Dateien auf dem Gerät des Nutzers, 2015
10. Patrick Gerken. Statische Sicherheitsanalyse von Java Enterprise-Anwendungen mittels Program Slicing, 2015
11. Florian Thomas. Modellierung von Informationen zur Interprozesskommunikation in Android-Anwendungen für Datenflussdiagramme, 2015
12. Sebastian Feldmann. Konzeption und Implementierung einer Eingabeprüfung für Struts-basierte Webanwendungen, 2015
13. Daniel Müller. Untersuchung zur Broadcast-Sicherheit in Android-Apps, 2015
14. Daniel Schwarz. Sicherheitsanalyse der clientseitigen Umsetzung des OAuth-Protokolls in Android-Anwendungen, 2016

15. Maximilian Schönborn. Detektion von Shared Preferences-Einträgen in Android-Applikationen mit Hilfe statische Programmanalyse, 2016
16. Jan Bartkowski. Sicherheitsanalyse eines App-gesteuerten Smart Home Systems, 2016
17. Mathias Detmers. Evaluation des WALA-Slicers bzgl. der Anwendbarkeit auf sicherheitskritische Java-Programme, 2016
18. Patrick Lorenz. Sicherheitsanalyse von Smarthome Android Apps, 2017
19. Paul Warsewa. Informationssicherheit für Laien, 2017
20. Timo Glander. Sicherheitsanalyse einer Android App zur entfernten Steuerung eines Industrial Controllers, 2017
21. Jonas Rahlf. Prüfung des korrekten Einsatzes von Krypto-APIs mit der Java Modeling Language und eines Extended Static Checkers, 2017
22. Kevin Skyba. Android Smarthome App Security Analysis, 2017
23. Luca Glockow. Sicherheitsanalyse einer Smarthome-Zentrale, 2017
24. Christoph Wohlers. Sicherheitsuntersuchung der Kommunikation eines FHSS-basierten Babyfons, 2018
25. Arian Mehrfard. Sicherheitsuntersuchung einer Android App für den Zugriff auf den Personalausweis, 2018
26. Tristan Bruns. Security Analysis of a Smart Home System: The Example of IKEA TRADFRI., 2018
27. Jerome Schmidt. Entwicklung eines Eingabeformates zur Definition von Securitypatterns für ein Sicherheitsanalysewerkzeug, 2018
28. Lorenz Hüther. Analyse von Datenerhebungsmethoden durch Smart-TVs, 2018
29. Jannis Ötjengerdes, Connor Lannigan. IT-Sicherheitsanalyse der Smarthome-Plattform openHAB2, 2018
30. Lasse Künzel. Sichere Vewendung der Qt-Bibliothek, 2018
31. Tobias Osmers. Sicherheitsanalyse der TLS-Client-Implementierung von Android-Anwendungen bezüglich ihrer Kommunikation mit einem Gerät im lokalen Netz, 2018
32. Jannis Fink. Low-Level Security Patterns for Android Apps controlling IoT device. 2019
33. Alim Kerimov. Evaluation eines auf Slicing basierenden Codeanalyse-Werkzeugs in Bezug auf seine praktische Anwendbarkeit im Kontext der IT-Sicherheit, 2019

34. Jasper Wiegratz. Sicherheitsgrundlagen von Docker-Images im Kontext der Softwareentwicklung mit DevOps am Beispiel eines Continuous Integration und Delivery Prozesses, 2019
35. Merlin Hanke. Sicherheitsanalyse des Instant-App-Konzept in dem Android-Betriebssystem, 2019

Master and Diploma Theses

1. Kim Schoen. Sichere Kommunikation in sporadischen Kundenbeziehungen, 2003
2. Daniela Bork. Sicherheitszertifizierung am Beispiel eines Marktplatzverbundes, 2003
3. Ersin Ürer. Untersuchung von WLAN-Sicherheitsprotokollen, 2005
4. Lars Migge. Spezifikation und Durchsetzung rollenbasierter Security Policies, 2005
5. Tanveer Mustafa. Design and Implementation of an Role-based Authorization Engine, 2006
6. Xinyu Bao, Yan Guo. Durchsetzung von organisatorischen Richtlinien in Web Services mit Hilfe von UML und OCL, 2007
7. Silke Schäfer. Konstruktion von sicheren RFID-Anwendungen, 2007
8. Adrian Nowak. Sicherheitsaspekte mobiler Endgeräte, 2007
9. Stefanie Gerdes. Rollenbasiertes Sicherheitskonzept für Krankenhäuser unter Berücksichtigung der aktuellen Entwicklungen in der Gesundheitstelematik, 2007
10. Meike Klose. Grundzüge eines IT-Sicherheitskonzeptes für Apotheken unter der Berücksichtigung der Gesundheitstelematik, 2008
11. Marc Ebler. Eine Sicherheitsanalyse zum Einsatz von mobilen Endgeräten im Außendienst, 2008
12. Assoulian Mkliwa Tchamsi. Umsetzung von dynamischen RBAC Policies mit Hilfe von UML und OCL, 2009
13. Raffael Rittmeier. Grundzüge eines Sicherheitskonzepts für Arztpraxen unter Berücksichtigung der Gesundheitstelematik, 2009
14. Jan Osmers. Ein Leitfaden für hohe Informationssicherheit beim mobilen Arbeiten, 2010

15. David Kamga Adamo. Entwicklung eines rollenbasierten Autorisierungswerkzeuges für Workflows auf Basis eines Modellprüfers, 2010

16. Florian Junge. Dynamische Erstellung von Angriffsbäumen für Rechnernetze mittels eines modularen Werkzeugs, 2010

17. Stefan Klement. Sicherheitsaspekte der Google Android Plattform, 2011

18. Bernd Samjeske. Entwicklung eines erweiterbaren ontologiebasierten Asset-Managements, 2012

19. Timo Reimerdes. Sicherheit und Privatsphäre in Sozialen Netzwerken, 2012

20. Bastian Breit. Sicherheitsaspekte von Android und mobilen Verkaufsportalen, 2013

21. Dimitri Hellmann. Angriffsszenarien ausgehend von Android-Anwendungen, 2013

22. Axel Auffarth. Modellierung von Sicherheitsaspekten in Softwarearchitekturen, 2013

23. Christian Liebig. Sicherheitsanalyse von mobilen Geschäftsanwendungen, 2014

24. Malte Humann. Auswirkungen von Sensoreigenschaften auf die Angriffserkennung mittels Sensorfusion, 2014

25. Oliver Schnieders. Identitätsmanagement im E-Commerce, 2014

26. Tim Schleier. Erstellung einer bidirektionellen Kommunikation mit CBOR als Datenformat, 2014

27. Markus Gulmann. Statische Sicherheitsanalyse der Android Systemservices, 2014

28. Katharina Hafner. Modellierung von Rollenkonzepten für Krankenhäuser mittels UML und OCL, 2015

29. Kevin Löhmann. Analyse und Beschreibung des Binder-Frameworks zur Interprozesskommunikation unter Android als Grundlage für weiterführende Sicherheitsbetrachtungen, 2015

30. Stefan Gommer. Identifikation von kritischen Informationsflüssen in Android-Anwendungen auf Basis von statischen Programmanalysen, 2015

31. Kai Hillmann. Sicherheitsanalyse von App-gesteuerten Alarmanlagen, 2015

32. Fritjof Bornebusch. New concept of the Android keystore service with regard to security and portability, 2016

33. Henning Ziegler. Analyse der Verwendung von Kryptographie-APIs in Java-basierten Anwendungen, 2016

34. Philipp Hirch. Automatische Inferenz von JML-Sicherheitsspezifikationen mit Exception Handling, 2016

35. Marcel Schuster. Modellierung und Validierung von Rechnernetzen auf unteren OSI-Schichten mit UML und OCL, 2017

36. Daniel Müller. Reference Security Guide for App-Controlled Smart Home Systems, 2017

37. Philipp Kolloge. Erweiterte Sicherheitsanalyse eines App-gesteuerten Smart Home Systems, 2018

38. Dario Treffenfeld-Mäder. Konzeption und prototypische Entwicklung einer Plattform zur Unterstützung des Programmverstehens der IT-Sicherheit von Anwendungen Handling, 2018

39. Philip Nguyen. Statische Sicherheitsanalyse mit automatisierten Code Audits - Sicherheitsanalyse von Java-Applikationen mit erweitertem Programm Slicing, 2017

40. Jörg Wilhelms. Sicherheitstechnische Untersuchung von Debug-Schnittstellen auf Android-basierten Smart-TVs und Smartphones, 2018

41. Jan Bartkowski. Evaluation der Machbarkeit von Threat Modeling und automatisierten Sicherheitstests für eine reale Cloud-Anwendung, 2018

42. Mathias Detmers. Sicherheitsanalyse der OSCI 1.2 Transport Bibliothek, 2019.

43. Michael Cyl. Sicherheitsanalyse für Android-Systemdienste auf der Basis von Program Slicing, 2019.

Appendix D

Acquisition of Third-Party Funding

Over the years, Dr. Karsten Sohr has acquired funding of about 5 million € from different funding bodies for the University of Bremen. He has contributed to many of these proposals, by either defining the entire project idea or by writing substantial parts of the proposal, including, ForRBAC (DFG), ORKA, fides, VOGUE, ASKS, SAiM, ZertApps, PortSec, SecureSmartHomeApp (all funded by the BMBF), CertifiedApplications (BMWi), and SecProPort (BMVI). For writing these proposals successfully, profound knowledge on the respective research areas was a prerequisite.

The following projects have been acquired:

1. SmartHomeDevices – Erprobung und Bewertung des Sicherheitsstandards Smart Home Devices, Contractor: Bundesamt für Sicherheit in der Informationsstechnik (BSI), Funding: 12,000 €, Period: 07/19 – 12/19

2. SecProPort – Skalierbare Sicherheitsarchitekturen für die Geschäftsprozesse in deutschen Häfen, Funding body: BMVI, Funding: 612,000 €, Period: 11/18 – 10/21

3. SecAnalysisOSCI – Begutachtung der OSCI-Bibliothek (Java), Contractor: Koordinierungsstelle IT-Standards des Bundes und der Länder, Funding: 17,000 €, Period: 11/17-10/18

4. SecureSmartHomeApp – Entwicklung sicherer mobiler Anwendungen zur Steuerung von Smarthome-Systemen, Funding body: BMBF, Funding: 255,000 €, Period: 01/17 – 12/18

5. PortSec – IT-Risikomanagement in der Hafentelematik auf Basis der Software-Architektur, Funding body: BMBF, Funding: 381,000 €, Period: 09/16 – 08/18

6. CertifiedApplications – Leichtgewichtige Sicherheitszertifizierung für Java-Anwendungen mittels werkzeuggestützter Programmanalysen, Funding body: BMWi, Funding: 190,000 €, Period: 05/16 – 04/18

7. SecPatterns – Erkennung und Validierung von Security Patterns, Funding: 170.000 €, Funding body: DFG, Period: 01/16 – 12/17

8. ZertApps – Zertifizierte Sicherheit für mobile Applikationen, Funding: 210,000 €, Funding body: BMBF, Period: 01/14 – 12/15

9. iMonitor – Entwicklung von optimierten und parallelen Inferenzverfahren für SIEM-Systeme, Funding: 175,000 €, Funding body: BMWi, Period:07/ 2013 – 06/2015

10. SAiM – Schutz Androids durch intelligentes Monitoring, Funding: 210,000 €, Funding body: BMBF, Period: 06/2013 – 05/2015

11. HafenSecurity – Architekturelle Sicherheitsanalyse einer Hafensoftware, Funding: 5,000 €, Contractor: dbh Logistics IT AG, Period: 03/2013

12. ASKS – Architekturbasierte Sicherheitsanalyse geschäftskritischer Software-Systeme, Funding: 217,000 €, Funding body: BMBF, Period: 07/2010 – 06/2012

13. VOGUE – Vertrauenswürdiger mobiler Zugriff auf Unternehmensnetze, Funding: 255,000 €, Funding body: BMBF, Period: 10/2009 – 09/2011

14. FIDeS – Frühwarn- und Intrusion Detection System auf Basis kombinierter Methoden der Künstlichen Intelligenz, Funding: 936,000 €, Funding body: BMBF, Period: 09/2008 – 02/2012

15. InTaSIEM – Indikator-Taxonomie zur Unterstützung des SIEM-Integrationsprozesses, Funding: 33,000 €, Contractor: T-Systems International GmbH, Period: 09/2010 – 12/2010

16. SiWear – Sichere Wearable-Systeme zur Kommissionierung industrieller Güter sowie für Diagnose, Wartung und Reparatur, Funding: 120,000 €, Funding body: BMWi, Period:09/ 2007 – 06/2010

17. SIMOIT – Sicherer Zugriff von mobilen Mitarbeitern auf die IT-Infrastruktur von mittelständisch geprägten Unternehmen, Funding: 33,000 €, Funding body Wirtschaftsförderung Bremen, 04/07 – 02/08

18. Mobiltelefon-Demonstrator – Demonstration einer Sicherheitslücke in Mobiltelefonen, Funding: 10,000 €, Contractor: Bundesamt für Sicherheit in der Informationstechnik, Period: 02/2008 – 06/2008

19. ORKA – Organisatorische Kontrollarchitektur: Der Weg von einem statischen Berechtigungsmanagement zu einem dynamischen, Funding: 230,000 €, Funding body BMBF, Period: 08/06 – 02/09

20. ForRBAC – Formale Spezifikation, Verifikation und Umsetzung von rollenbasierten Sicherheitsrichtlinien, Funding: 115,000 €, Funding body: DFG, Period: 11/06 – 11/08

21. RFIDSec – Erstellen einer Studie zum Thema „Technologiezentrierte Sicherheit in RFID-Systemen“, Funding: 17,500 €, Contractor: BMBF, Period: 09/2006 – 02/2007

22. ITSEC – Erstellen eines E-Learning-Portals für IT-Sicherheit, Funding: 46,000 €, Contractor: Institut für Wissenstransfer GmbH), Period: 03/2006 – 01/2007

23. KoSIT – Modellierung von Daten im E-Government mittels UML/OCL; Funding: 723,000 €, Contractor: Koordinierungsstelle IT-Standards des Bundes und der Länder, Period: 07/05 – 07/14

24. Live-Hack Demos – Durchführung von Live Hack-Demos und Penetrationstests, Funding: 29,000 €, Contractor: various, e.g., Datenschutzbeauftragter der Landkreise Diepholz und Delmenhorst, Nemetschek Bausoftware AG, Wirtschaftsförderung Bremen, ZF Friedrichshafen AG, BSAG, Period: 01/05 – today

Appendix E

Accumulated Publications

The following publications are part of this cumulative habilitation thesis:

1. K. Sohr, B. Berger. Idea: Towards Architecture-Centric Security Analysis of Software. Proc. 2nd International Symposium on Engineering Secure Software and Systems (ESSoS 2010), Pisa, Italy, 2010.

   Own contribution: Defining the general research direction of architectural risk analysis based on reverse engineering of (Java) software and writing large parts of this paper.

2. B. Berger, K. Sohr, R. Koschke: Extracting and Analyzing the Implemented Security Architecture of Business Applications. In Proc. of the 17th European Conference on Software Maintenance and Reengineering (CSMR 2013), Genua, Italy, 2013.

   Own Contribution: Defining the general research direction of reconstructing dataflow diagrams, which are widely used in the context of architectural security analyses, from Java-based software systems.

3. T. Mustafa, K. Sohr. Understanding the implemented access control policy of Android system services with slicing and extended static checking. International Journal of Information Security (IJIS). 14, 4 (August 2015), 347-366.

   Own contribution: Defining the idea of combining program slicing, DBC contracts as well as extended static checking for redocumentation of the implemented access control policy of the Android OS. Writing parts of this journal paper, e.g., introduction, basic technologies, and approach.

4. K. Sohr, M. Drouineaud, G.-J. Ahn, M. Gogolla. Analyzing and Managing Role-Based Access Control Policies. IEEE Transactions on Knowledge and Data Engineering, Vol. 20, No. 7, 2008.

   Own contribution: Describing the UML-based approach to specifying and validating role-based policies. Writing large parts of this paper.

5. M. Kuhlmann, K. Sohr, M. Gogolla. Comprehensive Two-Level Analysis of Static and Dynamic RBAC Constraints with UML and OCL. In Proc. 5th IEEE International Conference on Secure Software Integration and Reliability Improvement (SSIRI 11), Jeju Island, South Korea, June 2011. **Best Paper Award**.

   Own contribution: Defining the general research idea of the paper to specify and validate dynamic role-based policies, such as various variants of dynamic separation of duty, with the help of UML/OCL and accompanying UML tools. Presenting the paper at the SSIRI conference on Jeju Island, South Korea (Best Paper Award).

6. K. Sohr, M. Kuhlmann, M. Gogolla, H. Hu, G.-J. Ahn. Comprehensive Two-Level Analysis of Role-Based Delegation and Revocation Policies with UML and OCL. In Information and Software Technology (IST), Volume 54, No. 12, 2012.

   Own contribution: Defining the general idea of the paper to specify and validate delegation and revocation policies with the help of UML/OCL and accompanying UML tools. Writing the core parts of the paper concerning the specification approach for delegation policies based upon UML/OCL and the evaluation part.

7. M. Kuhlmann, K. Sohr, M. Gogolla. Employing UML and OCL for Designing and Analyzing Role-Based Access Control. In Mathematical Structures in Computer Science, Vol. 23, No. 4, 2013.

   Own contribution: Defining the general research idea of the paper as an extension of the SSIRI paper. Applying UML/OCL to *dynamic* role-based policies to a concrete case study, i.e., investigating those policies that include a kind of state (for example, History-based Separation of Duty or Dynamic Object-Based Separation of Duty).

8. M. Bunke, K. Sohr. Towards Supporting Software Assurance Assessments by Detecting Security Patterns. Software Quality Journal, Springer, 2020. To appear.

   Own contribution: Supervising the underlying PhD thesis of Dr. Michaela Bunke. Providing the general direction of this journal submission with a focus on applying the static analysis approach to detecting security patterns in Android applications. Defining parts of the comprehensive case study. Thoroughly revising the whole paper and writing parts of the discussion section.

9. B. Berger, C. Maeder, R. Wete Nguempnang, K. Sohr, C. Rubio-Medrano. Towards Effective Verification of Multi-Model Access Control Properties. In Proc. of the 2019 ACM Symposium on Access Control Models and Technologies (SACMAT), Toronto, Canada, June 2019.

   Own contribution: Defining the general research idea combining program analysis techniques for reverse engineering with UML-based modeling of role-based access control policies. Providing those role-based concepts that are supported by the presented approach.