Security-Pattern Recognition and Validation

Dissertation

Submitted by
Michaela Bunke

on
12th December 2018

to the
Universität Bremen
Faculty of Mathematics and Computer Science

in partial fulfillment of the requirements for the degree of
Doktor der Ingenieurwissenschaften
– Dr.-Ing. –

Defended
25th January 2019

Reviewed by
Prof. Dr. Hans-Jörg Kreowski
Universität Bremen, Germany

and
Dr. Karsten Sohr
Universität Bremen, Germany
In Memorial of

Ilse Schlamilch
★ 21 November 1924
† 09 June 2017

Karl Schlamilch
★ 03 March 1927
† 19 June 2018

Manfred Friedrichs
★ 29 August 1935
† 3 July 2017
ABSTRACT

The increasing and diverse number of technologies that are connected to the Internet, such as distributed enterprise systems or small electronic devices like smartphones, brings the topic IT security to the foreground. We interact daily with these technologies and spend much trust on a well-established software development process. However, security vulnerabilities appear in software on all kinds of PC(-like) platforms, and more and more vulnerabilities are published, which compromise systems and their users. Thus, software has also to be modified due to changing requirements, bugs, and security flaws and software engineers must more and more face security issues during the software design; especially maintenance programmers must deal with such use cases after a software has been released.

In the domain of software development, design patterns have been proposed as the best-known solutions for recurring problems in software design. Analogously, security patterns are best practices aiming at ensuring security. This thesis develops a deeper understanding of the nature of security patterns. It focuses on their validation and detection regarding the support of reviews and maintenance activities.

The landscape of security patterns is diverse. Thus, published security patterns are collected and organized to identify software-related security patterns. The description of the selected software-security patterns is assessed, and they are compared against the common design patterns described by Gamma et al. to identify differences and issues that may influence the detection of security patterns. Based on these insights and a manual detection approach, we illustrate an automatic detection method for security patterns. The approach is implemented in a tool and evaluated in a case study with 25 real-world Android applications from Google Play.
ZUSAMMENFASSUNG


ACKNOWLEDGEMENTS

The acknowledgments are not included in the electronic version for reasons of data protection.
# TABLE OF CONTENTS

## I  Prelude

### 1  Introduction
- 1.1 Motivation  
- 1.2 Problem Statement  
- 1.3 Approach
  - 1.3.1 Security Pattern Validation  
  - 1.3.2 Security Pattern Detection  
- 1.4 Contributions  
- 1.5 Origin of Chapters and Related Publications  
- 1.6 Thesis Outline

## II  Validation

### 3  Literature Review
- 3.1 Article Selection and Discovery  
  - 3.1.1 Process  
  - 3.1.2 Result  
- 3.2 Summary
6 Software-Security Patterns: Degree of Maturity 69
   6.1 Introduction .............................................. 69
   6.2 Patterns and the Software Life-Cycle ...................... 71
      6.2.1 Software Engineering—Forward Engineering ............ 72
      6.2.2 Software Maintenance—Reverse Engineering .......... 73
      6.2.3 Resumé .................................................. 74
   6.3 Possible Issues Affecting the Adoption of Patterns ....... 75
      6.3.1 Terminology ............................................ 75
      6.3.2 Classification ......................................... 76
      6.3.3 Pattern Presentation .................................... 77
      6.3.4 Resumé .................................................. 83
   6.4 Summary ..................................................... 83

III Recognition 85

7 Manual Security-Pattern Detection 87
   7.1 Bauhaus Tool and Hierarchical Reflexion Analysis ....... 87
   7.2 Early Case Studies ......................................... 88
      7.2.1 Single Access Point Pattern .......................... 89
      7.2.2 Case Study: Spark .................................... 90
      7.2.3 Case Study: Simple Android Instant Messaging Application . 91
      7.2.4 Resumé .................................................. 92
   7.3 Security Aspects and the RFG .............................. 92
      7.3.1 Security Patterns at the Architecture Level .......... 93
      7.3.2 Automatic Detection, Suggestions, and Learning ...... 93
      7.3.3 Source and Sink Markers for Pattern Endings ....... 93
   7.4 Summary ..................................................... 94

8 Security Patterns in Source Code 95
   8.1 Discovering Patterns and Micro-Architectures ............ 96
      8.1.1 Mining Patterns ........................................ 96
      8.1.2 Discovering Micro-Architectures based on Patterns .. 98
   8.2 Security Patterns and Android Security .................... 99
      8.2.1 Patterns Relevant to Securing Android Apps .......... 100
      8.2.2 Collecting Security Guidelines ........................ 100
      8.2.3 Relating Patterns and Guidelines ..................... 101
      8.2.4 Selected Security Patterns ............................. 101
   8.3 Security-Pattern Variants ................................. 102
      8.3.1 Authentication Enforcer ............................... 102
      8.3.2 Information Obscurity ................................ 106
      8.3.3 Secure Pipe ............................................ 109
      8.3.4 Result .................................................. 114
   8.4 Summary ..................................................... 114
9 Security-Pattern Detection Tool

9.1 Introduction .................................................. 117
9.2 Motivation ..................................................... 118
9.3 OPG Concepts and Enhancements for COPGs .................. 120
  9.3.1 Object Process Graph (OPG) .......................... 121
  9.3.2 Connected Object Process Graph (COPG) .......... 122
9.4 Analysis Tool ................................................ 126
  9.4.1 Preprocessing ......................................... 127
  9.4.2 Normal Soot Analyses ................................ 133
  9.4.3 Identification of Security Patterns .................... 133
9.5 Summary ....................................................... 134

10 Tool Evaluation ................................................. 135

10.1 Case Study ................................................... 135
  10.1.1 Defining and Validating the Hypothesis ............ 135
  10.1.2 Method of Comparison ................................ 136
  10.1.3 Selecting Apps ...................................... 137
  10.1.4 Analyzing the Results ............................... 140
10.2 Exemplary Usage of COPGs ................................ 144
10.3 Discussion ..................................................... 146
  10.3.1 Usage Scenarios .................................... 146
  10.3.2 False Positives and Negatives ....................... 147
10.4 Summary ....................................................... 147

IV Finale .......................................................... 149

11 Related Work .................................................. 151

11.1 Security Patterns ........................................... 151
  11.1.1 Collection ........................................... 151
  11.1.2 Classification ....................................... 152
  11.1.3 Description Quality ................................. 152
11.2 Pattern Detection ........................................... 153
  11.2.1 Design Patterns .................................... 153
  11.2.2 Security Patterns ................................. 155
11.3 Detecting Security Concepts at Architectural Level ......... 155
11.4 Related Techniques ......................................... 156
  11.4.1 Static Trace Extraction ............................. 156
  11.4.2 Slicing .............................................. 157
  11.4.3 Type-State Analysis ............................... 158
  11.4.4 Feature Location .................................. 159
  11.4.5 Diagram Extraction ............................... 161
11.5 Static Security Analysis for Android ....................... 162
  11.5.1 Modeling Android-Specific Concepts ............... 162
  11.5.2 Slicing .............................................. 163
Part I

Prelude
Nowadays, computers are omnipresent. They can be found in nearly every part of our daily life, e.g., tablets, smartphones, TVs, and cars. An increasing and diverse number of these technologies is connected to the Internet such as distributed enterprise systems or small electronic devices like the iPad. More or less, all of these devices are running different types of software that should exhibit — depending on their service — security features and thereby fulfill context-dependent security assurances. For example, on a smartphone, a user expects that a home-banking application encrypts the communication between the bank and the phone to ensure security properties such as integrity and confidentiality which brings the topic IT security to the foreground.

We interact daily with the computer devices mentioned above and spend much trust on a well-established software development process. Several techniques have been developed to provide security by design which means that software is designed with security in mind from the beginning, such as threat modeling [472], security architecture design [179, 272], security design guidelines, security design principles [416], security patterns [538], and security design review [326] (Figure 1.1). These techniques are used to make sure that security is built into a software system before it will be implemented to minimize impact when a security vulnerability is discovered. Besides, activities such as security reviews are suggested to assess the security assurances with regard to governmental directives or certain requirements to ensure that security is correctly built in before releasing software to the public or customer [333].

However, security vulnerabilities appear in the software on all kind of PC(-like) platforms [352, 393, 482, 493]. Especially, the development of secure mobile applications seems to be quite difficult as published vulnerabilities in mobile apps indicate [83, 123, 134, 139, 493]. Hence, developers must more and more face security issues during software design and especially reengineering.
1.1 Motivation

As mentioned before software should be developed with security by design and has to be modified due to bugs and security flaws. Thus, security gains more and more attention in the area of software development, and maintenance programmers and reviewers (analysts) are progressively faced with implementing and applying security solutions to their software system. They must check the code quality and implemented security features of software systems or fix programmed security features in the code. Among others, one challenge they face is to understand the whole application and the implemented software-security aspects and have to focus on separating correct from faulty implementations. Today’s security tools can provide basic support during review activities based on static analysis \[42, 325, 332, 485\]. Chess et al. describe in depth how static analysis tools can detect bugs in code such as buffer overflows or missing input validation \[81\]. However, as indicated above, such analyses detect low-level implementation bugs, which often only manifest in one line of code.

Although useful, current static analysis tools do not help in finding application-specific defects, where a component is not used according to its specification or security function. Analysts must understand the general idea/function of a security feature, e.g., how to secure a client-server communication with encryption, as a prerequisite to inspecting and understanding implemented software-security aspects. Nevertheless, knowing security features and their code locations is not enough to separate a correct from a faulty implementation in the source code. It has been observed that one of the challenges in securing applications is to configure the security-related framework objects in the right way \[32, 139\].

Figure 1.2 depicts the challenges an analyst has to face on assessing security features in the code. To answer question Q1, the analyst must have documentation or experience with the software to be analyzed to find code areas that can implement the wanted security feature. Identifying these starting points is often based on experience rather than on documentation. Moreover, she has to know that function this security feature has to fulfill and how it is integrated into the software design (question Q2). Consider question Q3 here the analyst must know which combinations of (Java) objects form a specific security feature. Therefore, the analyst must have
good knowledge of security mechanisms and features in general, but also much experience on how they must be implemented in code. The analyst can answer question Q4 by searching and tracing manually identified objects in the source code. Finally, the analyst can answer question Q5 by assembling the priorly collected (code) information and focus on the relevant interacting objects to decide whether their configuration is correct or faulty to achieve the desired security requirement.

The comprehension tasks formulated in the questions Q1, Q2, Q3, and Q4 can be supported by detecting security patterns [503] in code analogously to design patterns [194]. Security patterns provide reusable solutions to recurring security problems and describe participants and their interaction to ensure a specific (security) property [538]. To help the analyst addressing question Q5, she needs a tool that visualizes an object configuration, and its interaction points with other (Java) objects that may influence its setup or functionality. Assisted by an object configuration visualization, an analyst can answer question Q6 more easily by having a security pattern’s general goal and the reviewed implementation in mind. Moreover, the software architecture documentation can be validated and possibly completed by adding detected security patterns.

Given that patterns, in general, are the best-known solution to a recurring problem [173], security patterns should also be recognized during the maintenance process to guarantee security objectives and requirements. For example, a detected Check Point pattern [437] allows an analyst to conclude that at this point in code relevant security information, e.g., credentials, will be validated before further steps in the application flow are carried out. This knowledge of the recognized pattern allows the programmer to obtain the original design and the associated functionality of the component during maintenance. In this specific case, the developer can avoid bypassing this Check Point in further implementations. In addition, a reviewer can use a detected security pattern to determine whether the software meets certain security requirements and, if necessary, initiate maintenance activities to increase the security in the program.
1.2 Problem Statement

Currently, security patterns are only considered and used in software designing activities but not in software reviews or reengineering activities such as maintenance programming. So far, it is insufficiently clarified whether the definition of security patterns in literature is sufficient for the recognition of security patterns.

In 1997 Yoder and Barcalow published the first patterns targeting security under the topic of security patterns [538]. At that time, their popularity in the pattern community grew and many patterns have been published after that [238]. Until the publication of the three main works on security patterns [144, 437, 464] various works have been published with security patterns and security-pattern catalogs [274, 282, 401, 403, 439, 540]. The latest collection was published in 2013 with 97 security patterns [205], and the collection summarizing the most security patterns (180) was published in 2009 [439]. Due to that fact, the existing security pattern catalogs seem to be quite incomplete. Thus, security patterns should be collected to have a viable base for further inspections.

Like design patterns, which are also called Gang-of-Four (GoF) patterns [173], security patterns can be applied to implementing and structuring software systems, however, they can also model security issues and security processes in enterprises such as “Enterprise Architecture Management Patterns” [136]. Based on this heterogeneity, the classification of security patterns is an important and often discussed issue [206, 209, 437, 516]. However, the existing classification approaches for security patterns often imply that the patterns to be organized belong to a specific domain, e.g., software. In addition, most of these classifications only consider a few patterns for their organizing process, e.g., the seven patterns presented by Yoder and Barcalow.

The POSA model described by Buschmann et al. [72] is said to be frequently used to describe the context and usage of security patterns. However, within the pattern community, various descriptive models for security patterns exist [437]. Some descriptions make use of UML diagrams or rarely source code snippets to clarify the security pattern modeling during the design phase. Nevertheless, security patterns are mostly described highly abstract; so it is difficult to understand the benefit or use if one is not familiar with software design linked with security issues [238]. Those circumstances—among others [103]—have an impact on the software design when one has to ensure security objectives and select an appropriate security pattern for the software needs. These issues can also be problematic for their recognition within software systems.

Security patterns also seem to differ from Gamma’s design patterns which are already used for software design redocumentation. VanHilst and Fernandez looked into that subject whether existing detection techniques for design patterns can be applied to security patterns [503]. They summarized their results with the statement “GoF Patterns are not security patterns” [503]. They postulate the following differences between design and security patterns:

1. Security patterns are larger than GoF patterns.
(2) Security patterns tend to be abstract and allow more variability in the implementation than GoF patterns.

(3) Security patterns always appear in tight relationships with other security patterns.

Furthermore, complexity as a difference is possible. Hence, security patterns have similarities with design patterns, but cannot be treated the same at detection. For example, by securing communication, two applications (client/server) have different roles and implementations of the implemented pattern so that they must be separated in an analysis. Therefore, it is necessary to describe them in greater detail before planning their detection.

For maintenance and review activities, several reengineering tools exist to obtain a clear view of software structure and behavior. However, only a few of those tools take the design patterns into account to support program comprehension at that point. VanHilst and Fernandez have listed some approaches [503]. As far as we know, only one approach has been presented that supports the detection of security patterns [16]. However, this approach is limited to patterns providing class and sequence diagrams and does not focus on supporting analysts in inspecting a code base.

In a final analysis, the shown factors do not support the application, comprehension, and recognition of security patterns in software. Therefore, it is desirable to collect and validate the existing security patterns and develop a recognition tool. Such a tool can ensure that security patterns are detected and preserved during the software maintenance process or assessed in a review.

1.3 Approach

The approach presented in this thesis is two-fold. On the one hand, the nature of security patterns is evaluated, and on the other hand, a viable detection approach is developed with respect to the priorly detected issues.

1.3.1 Security Pattern Validation

This thesis part applies to the problem stated by Shostack [447]: "However, in practice, these patterns have not been popular. The reasons for this are not clear, and those investing in using patterns to address security problems would likely benefit from studying the factors that have limited their popularity." This thesis part addresses the aforementioned problem by dealing with some topics that are already marked to have possibly negative influences on the adoption of security patterns [103]. So far, the research in the area of security patterns is limited to the software design phase; moreover, maintenance and review aspects are left unattended. Thus, this work focuses on these issues from an analyst’s point of view.

Some works mention that the range of described security patterns is wide and a comprehensive collection of security patterns is an open problem within the security
pattern community [103, 238]. For example, Hafiz et al. have collected 97 security patterns but use only 14 patterns for their organizing approach [209]. Thus, security patterns published from 1997 to 2016 are collected and a classification of security patterns is developed. This organizing approach is based on all collected patterns and enables the selection of patterns relevant to the pattern detection approach of this thesis. Furthermore, the collected security patterns serve as a base set for all further investigations within this thesis.

As mentioned before, it is supposed that design patterns have some aspects in common but also have differences with security patterns. Which differences they have has not been worked out so far. Therefore, the assumptions of VanHilst and Fernandez about the differences between design and security patterns [503] are addressed in this thesis by inspecting the description form, the classification and the usage of diagrams and code examples of security patterns. Moreover, the state-of-the-art of design and software-security patterns are surveyed with regard to the usage in the software life-cycle. All of these topics are of interest in applying software-security patterns and give clues for the pattern-detection approach. Huge differences between design and security patterns may demand the consideration of new pattern detection strategies compared to existing design-pattern approaches, and smaller differences may indicate the possibility of adapting existing approaches. All in all, this sharpens the scope of design and security patterns.

1.3.2 Security Pattern Detection

The usage of security patterns within software development has been demonstrated in a few studies [202, 208, 277]. Unfortunately, they examine only very few software systems and do not provide information which security patterns are used in fact to engineer software systems. Thus, we determine which security patterns are relevant to software engineering. This step allows us to prioritize the important security patterns within this thesis.

In line with the architectural analysis is the detection of existing security patterns of interest. Recognized patterns can support software architecture reconstruction and program comprehension. It facilitates the assessment of software with regard to several security requirements. Thus, we consider the detection at the architectural level as well as the source code level by enhancing the Object Process Graph (OPG) concept to deal with multiple-object interaction.

Since the description of security patterns is often abstract, we extract and validate micro-architectures of frequently-used security patterns. Based on that, we develop a tool that allows a semi-automatic security evaluation where the analyst is pointed to prominent structures. Moreover, the tool is evaluated with 25 publicly available real-world apps from Google Play and discussed with software-security experts of the SAFECode organization to obtain feedback on the need for connected objects and software comprehension to assess security features [190, 455].
1.4 Contributions

The contributions of this thesis are:

*Collection of security patterns* A systematic literature review is conducted to collect the published security patterns in the period of 1997 to 2016.

*Classification of security patterns* This thesis proposes two new classification schemes: a) The first summarizes all collected security patterns and organizes them into application domains and b) The second shows in detail which security and implementation nature security patterns with respect to software have.

*Analysis of software-security pattern descriptions* The descriptions of software-security patterns are analyzed and compared against the description templates of common design patterns to identify security-specific sections. Moreover, a section mapping is developed and tested with multiply described security patterns. This section mapping can help to deal with the different description forms of security patterns.

*Clarification of differences between design and security patterns* We compare design and security patterns to find indicators for negative impact on security pattern engineering in software development. The detected degree of maturity of security patterns is compared to common design patterns, and research opportunities on security patterns are depicted.

*Manifestation of security patterns in code* Also, pattern variants for three security patterns are extracted of the Android Framework API and described as Connected Object Process Graphs (COPGs).

*Detection of security patterns* We present two approaches to detecting security patterns. The first approach uses an existing program comprehension tool at the architectural level to find patterns manually. The gained experiences are used to solve issues for the second (automatic) detection approach based on Connected Object Process Graphs. The feasibility of this approach is shown in a case study that inspects 25 real-world Android applications and measures the precision and recall of the developed detection approach.

1.5 Origin of Chapters and Related Publications

Various parts of this thesis have been previously published. Table 1.1 lists the publications that constitute the contents of this thesis. For each publication, the chapters indicate where the respective content is located within this thesis. The publications are given in chronological order. The inspected period of time for the content of the Chapter 3, 4, 5, 6, and 8 has been extended from 1997 to 2016 (priorly 1997 to mid-2012) and the findings have been updated.
Table 1.1 – Published and submitted papers covering content of this thesis.

Furthermore, more research results have been published. They are not explicitly included in this thesis, but are related to it (Table 1.2).

Table 1.2 – Other publications not included in this thesis.

1.6 Thesis Outline

This thesis is organized into four parts: prelude, validation, recognition, and finale. The chapters within the validation and recognition parts contain the main
contributions of this thesis. The remainder of this work is structured as follows. The next chapter (Chapter 2) provides background information on topics related to this thesis, such as software security, software maintenance, program comprehension, and security patterns. Within the validation part, Chapter 3 describes a literature survey on security patterns published from 1997 to 2016. The collected patterns are organized in Chapter 4. A part of the result of this chapter—the collected software-security patterns—is used as the base for the pattern description analysis in Chapter 5 and to estimate the maturity of security patterns compared to the common design patterns of Gamma et al. [173] in Chapter 6. In the recognition part, Chapter 7 provides an early case study on detecting security patterns manually to identify possible problems. Based on the priorly identified problems, Chapter 8 illustrates how security patterns can be found in the source code and assess their differences to design patterns with regard to their automatic detection. Finally, Chapter 9 describes an automatic pattern detection approach based on Connected Object Process Graphs which is evaluated in Chapter 10. This thesis is closed with Chapter 11 providing an overview of related work and the conclusion in Chapter 12.
This chapter provides background information concerning the main topics of this thesis, namely software security, software maintenance as well as program comprehension (Figure 2.1). In the following, each of these topics is treated in a general manner and their interplay is discussed.

Figure 2.1 – Topics related to the subject of this thesis.

2.1 Software Maintenance

Software maintenance is the modification of a software product after delivery where faults are corrected, the performance improved or the product adapted to the changed environment needs [257]. Some studies show that software maintenance activities contribute between 50% and 80% of a product’s lifecycle cost [135, 307, 354]. Unfortunately, software developers, who usually perform software maintenance tasks, rarely had any training in software maintenance [454]. Also, they often have not participated in the software’s development from the beginning. Thus, it is not surprising that some other studies have shown that about 50% of the time in software maintenance is spent on understanding the code before a change can be applied [167].
2.2 Software Reengineering

Software engineering is a combination of usual software development activities, called *forward engineering* and its inversion called *reverse engineering* or *reengineering* (see Figure 2.2). The goal of reverse engineering is to recover unavailable documentation by raising the abstraction level, from code to design to requirements. This is often done in two ways: observing the running system (dynamic analysis) or analyzing the code (static analysis). The software reengineering discipline aims to support developers with several methods and techniques at maintenance tasks. Its objective is to understand the existing functionality (specification, design, implementation) and then to re-implement it to improve the system’s functionality or quality.

![Figure 2.2 – Software (re-)engineering activities with some security touchpoints.](image)

2.3 Software Security

Due to the rising number of vulnerabilities software security is an emerging area in software development [482]. Many detected security bugs can be used to exploit software and may harm the user’s data or compromise services used or provided by the software. For that reason, applications that are used in our daily life, such as smartphone applications, are of special interest in ensuring security and privacy properties.

2.3.1 Security by Design

Since literature states that if a change is applied late to the design of an application costs will arise [372], Methods have been developed to decrease the number of design-level flaws at an early stage of software development. Some of them are highlighted in Figure 2.2. For example, Microsoft’s threat modeling [472] or the architectural risk analysis proposed by McGraw [324] should help to discover security problems already during the software design phase. In academia, more formal approaches dealing with software security have also been established, notably, language-based security [412], model-driven security [88], and stepwise refinement [321].
2.3.2 Security Bug Finder

Besides the methods mentioned above at the architectural level, the existing security testing tools are mostly based on static code analyses and aim to provide support during the implementation phase. A range of academic and commercial static analysis tools have been developed attempting to detect potential security-related programming bugs such as buffer overflows, simple race conditions or a diverse number of injection vulnerabilities based on missing input validation [81]. All these tools have in common that their goal is to find security vulnerabilities based on erroneous coding. The presentation of the results within these tools is consequently limited to several categorized lists of findings, aggregated dashboard-like views, or execution traces leading to the identified vulnerabilities. Employing such analysis tools can increase the software’s security by ensuring its stability by filtering out bugs at code level as it is restricted to common bug classes. Certainly, it is equally important in maintenance and reviews to understand the security mechanisms of systems at a more abstract level of design [32, 294].

2.3.3 Software Security-Review

Static analysis tools are also suggested to be used in review activities during a software’s forward-engineering process [325, 332, 485]. Such reviews can have two goals. One is the detection of coding vulnerabilities, and the other is the identification of software-security assurances already implemented in the code. Software-security assurance determines the degree of confidence at which the software functions are implemented in the intended manner and are trustworthy [92]. Thereby, it is checked whether the software fulfills the required security or legal privacy requirements as defined by the applicable law such as the General Data Protection Regulation (Regulation (EU) 2016/679) directive [138]. Section 1.1 already presented some issues of such a review. Extensive security reviews and code assessments are also performed to gain advances in levels of a Common Criteria certification [348].

Nowadays, the level of security is often estimated by the absence of vulnerabilities [32, 128, 294]. Possible vulnerabilities are often detected by the aforementioned static analysis tools in the first place and then assessed according to relevance by a reviewer. However, reviewers are rarely security experts [32, 502], which increases the complexity of the code review process [32, 294]. Thus, a correctly or partly correct implemented security features are often not assessed due to a lack of information, lack of time, or lack of experience.

2.4 Program Comprehension

Software maintenance and review activities require program comprehension abilities. On comprehending software an analyst “must examine both the structural aspect of the source code (e.g., programming language syntax) and the nature of the problem domain (e.g., comments, documentation, and variable names) to extract
the information needed to fully understand any part of a software system” [319]. She benefits from tools that support program comprehension. For example, a general-purpose program comprehension tool like the Fujaba tool suite is useful to better understand large code bases [349, 499]. Such tools often provide clustering or filtering functions for finding and selecting software parts of interest more easily. In addition, some program comprehension tools use reverse engineering techniques to provide up-to-date information about the system, for example, for the detection of implemented design patterns [366, 512] or architecture reconstruction [510]. However, to support software analysts to understand the security risks of their code, it is argued that program-comprehension tools have to be designed especially for security purposes to fit the needs of security analysts ([324, page 125]).

2.5 Patterns

Patterns capture solutions that have developed and evolved. They are very common, and we have them everywhere around us, such as the leaves of plants, workflows, or colored patterns on t-shirts. The specialties of patterns with regard to software development are presented in the following sections.

2.5.1 Origins of the Design Pattern Technique

The architect Christopher Alexander was the first one who originated patterns as an architectural concept in the late 1970s [11]. Alexander realized that some common components, he used to design buildings, had been used repeatedly during his work to solve a particular design problem. Thus, he identified them, defined the problem they solved and described the solution they provide. Then, each time he had to solve a design problem he tried to apply one of his identified design patterns. This technique provides accuracy and consistency in the design. Alexander illustrated his ideas with the concept of design patterns in 1975 [10], two years later his pattern language for building [11], and described his design pattern technique finally in 1979 [9].

The main advantages of using patterns are the following:

1. The solution is considered to be good because it has already been applied multiple times before (time-tested).

2. The advantages and disadvantages of using a pattern are known in advance and can be taken into account during design activities.

3. The communication between different stakeholders (e.g., between the house architect and the building company) can be eased by using the vocabulary for design concepts established through patterns.
2.5.2 Software Design Patterns

Years later, based on Alexander’s described pattern technique software architects saw parallels between the described physical architecture and software design issues. In 1987, Kent Beck and Ward Cunningham began with shaping patterns for programming graphical user interfaces in Smalltalk [37]. Ultimately, design patterns became popular in Computer Science after Gamma et al. had published their book on design patterns in 1994 [173]. These (GoF) patterns are generally considered the foundation for all other patterns in computer science. Their patterns capture simple and elegant solutions to specific problems in object-oriented software design. Moreover, their book established design patterns as a method for designing software architectures.

After that work, many books have been published covering patterns applicable to specific programming languages, domains or frameworks. For example, Buschmann et al. started in 1996 a book-series presenting patterns for software architecture [72–74, 283, 430]. Moreover, new conferences like the Pattern Languages of Programs Conference (PLoP) focused on this emerging topic.

Since then, software architects and designers use patterns to solve well-known and recurrent problems during the software design phase (Figure 2.2). Due to that fact, two years later pattern-detection approaches tried to uncover used patterns in source code to support the program comprehension task of maintenance programmers [293]. Design pattern detection is an active research area, and it has been found out that the detection of design patterns can support the program comprehension of software systems [194]. More specific insights on software design-patterns are described in Chapter 5 and 6.

2.5.3 Security Patterns

Security patterns were first mentioned three years after the GoF design patterns in 1997. Yoder and Barcalow summarized some existing patterns under this new topic [538]. At that time, their popularity in the pattern community grew and many patterns have been published after that [238]. Yoshioka et al. [539] and Heyman et al. [238] surveyed the published security patterns till 2008.

The primary goal of security patterns is to harden software or infrastructures against common attacks and misuse. Thus, they also model security issues driven by the (software’s) requirements and are said to improve the built-in security in a (software) system at the (software) design stage [207, 214]. Similar to the design patterns introduced by Gamma et al. [173], security patterns describe a general reusable solution to a well-known problem based on best practice. Thus, implemented security patterns can be seen as a lead that a software system is (correctly) secured. Unfortunately, there is no such thing as a self-checking tool for security patterns. Thus, the correctness of an implemented security feature has to be checked manually by analysts.
2.6 Conclusion

As indicated by the number of published vulnerabilities [352, 482], we conclude that software security is a current topic. Many bugs and security problems are detected in the field after a software has been released [352]. Security is often considered as an afterthought in the software life-cycle [434] and often belongs to the field of software maintenance. Moreover, analysts are also often not skilled or familiar with security issues [32, 502]. Thereby, maintenance programmers and reviewers (analysts) are faced with implementing and understanding security features in software systems.

Due to lack of documentation, lack of information, lack of time, or other reasons, maintenance and review tasks such as eliminating security defects can be quite challenging for maintenance developers and extremely costly for a software vendor. Unfortunately, these analysts are often not security experts. Thus, the program comprehension way of looking at a problem can be used to support the security awareness of people who are involved in the software development lifecycle. This implies that techniques helping analysts to understand (parts of) a software better can reduce maintenance costs. Specifically, with regard to security, they are faced to understand the implemented software-security aspects and to separate correct and faulty implementations and fix them without creating further security defects or side effects. Concluding, analysts can benefit in their activities of well-suited program comprehension tools to produce better results.

Design patterns have emerged as a popular technique for (among others) designing software systems. Also with the specific target security patterns have been proposed. However, only a few tools take the well-known design patterns [173] into account to support program comprehension [503]. Presently, none of them supports the detection of security patterns, but ensuring security is a significant task. Security patterns should be recognized so that security patterns can be preserved during the software maintenance process and reviewing activities. Their highlighting allows one to well-directed implementations of new (security) features in software or their assessment in correlation to the software’s requirements.
Part II

Validation
CHAPTER THREE

LITERATURE REVIEW

Existing security pattern investigations are biased by their focus on a small set of security patterns, for example, the studies by Hafiz et al. [209], Konrad et al. [291], or Laverdiere et al. [300]. Providing a holistic and comprehensive collection of security patterns is an open problem within the security pattern community [103, 238]. For example, the SecurityPatterns.org website, which provides a list of security patterns, offers a few more patterns than the ones published by Yoder and Barcalow [538], but mixed with articles that describe the application of security patterns [439]. Besides that, three books [144, 437, 464] and the documents of Kienzle et al. [282], Yskout et al. [540], and Dougherty et al. [115] exist, which summarize some security patterns into a catalogue. Other published security patterns are scattered across academic literature mostly published at conferences. Therefore, this thesis starts with a literature survey that covers the whole range of published security patterns in the period of 1997 to 2016. The collected patterns are the base for further parts of this thesis.

3.1 Article Selection and Discovery

This section describes the systematically conducted literature survey collecting various security patterns. The process is oriented towards the guideline proposed by Kitchenham et al. [284, 285].

The literature research started with three security patterns books [144, 437, 464] and the surveys carried out by Laverdiere et al. [300], Heyman et al. [238], and Yoshioka et al. [539]. The surveys give a good overview of published security patterns, but they refer only to often described patterns. Hence, common pattern-related conferences were also considered (Table 3.1) and searched for security patterns in the IEEE Digital Library [256] and the ACM Digital Library [1].
3.1.1 Process

The security pattern collection process was split into two parts: article discovery and article inspection process (Figure 3.1).

![Article discovery process](image1)

(a) Article discovery process.

![Article inspection process](image2)

(b) Article inspection process.

**Figure 3.1** – Article discovery and inspection process.

**Article Discovery**

Based on the titles of the three security pattern books and the mentioned pattern publications in the surveys, several keywords that may indicate security pattern publications are selected, such as “cryptographic”, “security”, “software”, “secure”, and “pattern” (Figure 3.1a).

The discovery process was split into two parts, conferences, and digital libraries.
Conferences The published works between the years 1997 to 2016 of the aforementioned pattern conferences (Table 3.1) were skimmed for the selected keywords. At first, all publications that contain these keywords were selected. In this initial selection phase, 1815 articles were found. Articles matching some of the keywords were collected for a further inspection.

Digital Libraries On searching for other electronic publications the two digital libraries provided by IEEE and ACM were used [1, 256]. Both offer an extensive database search for publications. First of all, the simple search was used to find publications that contain the keywords mentioned above. Due to the fact that the number of results was very high and too unspecific, the advanced search field provided by the websites was used to obtain more localized results. There, the following options were concatenated with ”and” to obtain better results:

- The year of the publication date has been limited from the year of the first published security patterns 1997 to 2016.
- The full text must contain the word “pattern”.
- The title must contain one of the keywords mentioned above.

Unfortunately, these restrictions still provided many undesirable results. Therefore, we skimmed the result list from top to bottom—where the search engines of the digital libraries provided the ordering based on relevance—and discontinued if more than ten papers did not deal with security patterns in their abstract. All relevant articles were collected for the following inspection process.

Article Inspection

All collected articles were inspected according to their content to ensure that the publication described one or more security patterns (Figure 3.1b). Therefore, the abstracts were read to find out whether they described a security pattern and then we made a note of the authors, publication year, and title. After that, publications that had not been filtered out initially by its abstract were read to confirm that they describe security patterns. In this step, the list was enhanced with each identified pattern for further readings.

Finally, the publication’s references were scanned and referenced publications containing the keywords above were collected. Each collected cross-reference publication was checked whether it was not already contained in the final collection of publications and added to the preliminary collected article list to be inspected. The step of cross-reference scanning was stopped when no new publications containing the keywords were found.

3.1.2 Result

On collecting the security patterns diverse results were observed which are depicted in depth in the following sections.
Problems during the Discovery Process

The search in the ACM and IEEE Digital Library produced many false-positive articles that contained no security pattern descriptions, at a closer look. Instead, they deal with other (software) security aspects, such as discussing secure software design in practice [327]. Besides, some pattern conferences published their conference proceedings at the Digital Libraries. For this reason, some Digital Library results overlapped with the collected contributions of the pattern conferences.

Another problem was that two conferences allowed contributions written in the foreign language of the region. 54 contributions at SugarLoafPLoP conferences were written in Portuguese or Spanish and 32 contributions were written in Japanese at AsianPLoP conferences. Thereby, these contributions could not be inspected whether they describe security patterns.

Figure 3.2 – Article selection and discovery results.
3.1 – Article Selection and Discovery

Venue Distribution of Security Patterns

We observed that security patterns are only a fraction of the work done in the pattern community. The domain of the other published patterns is very manifold (e.g., software, learning, teaching, cooking). Moreover, pattern mining and pattern application are two other often addressed topics.

91 different publications describing security patterns were identified, including books, proceedings, and technical reports. Most of them were found by looking at The Hillside Group [483] pattern conferences, such as PLoP and EuroPLoP (Figure 3.2a). Measured on the total number of publications of the initially selected conferences 6.7% (PLoP) to 0.8% (PATTERNS) publications are describing security patterns. An exception is the KoalaPLoP which only took place two times and has no publication describing security patterns.

Other publication types containing many security patterns were books and technical reports, which had been discovered by cross-references. Cross-references also discovered the following new conferences:

- International Workshop on Security, Trust and Privacy in Grid Systems (GRID-STP)
- International Conference on Internet Computing (ICOMP)
- Working Conference on Data and Applications Security (DBSec)
- IEEE International Workshop on Security Engineering Environment (IWSEE)
- International Conference on Software Engineering and Knowledge Engineering (SEKE)
- International Conference on Database and Expert Systems Applications (DEXA)

Compared to the Hillside Group pattern conferences [483], the conferences mentioned above have only a few security-pattern publications. The majority of security patterns is published at EuroPLoP and PLoP conferences followed by books and technical reports (Figure 3.2b).

One-third of the publications depict only one pattern. 16 publications describe two and 14 publications present three patterns. In the last third, a publication describes four to 54 security patterns. In total, we found 528 security patterns.

Security Pattern Authors

On collecting the security patterns, some authors appear quite often as an author of security patterns. In total 146 people participated in writing the collected patterns. As Figure 3.3 indicates, Fernandez (co-)authored the most security pattern publications (42 of 91), followed by Larrondo-Petrie (eight publications),
Schumacher and Hashizume (five publications), Romanosky, Delessy, and Hafiz (four publications) and Ajaj and Monge (three publications). At least, 23 of the remaining authors were involved in two contributions. The remaining 113 authors only participated in one security pattern publication. The research of Fernandez and his 72 co-authors dominate the security-pattern research-community with regard to described security patterns (Figure 3.4). Such domination in a research area can be critical to the scientific objectivity. It can decrease the scientific objectivity through lesser scientists’ open discourse and not be sharing the intellectual authority equally among qualified practitioners [387].

Figure 3.3 – This tag cloud depicts the amount of published publications per author. A larger name indicates more, a smaller less publications per author.

Multiply-Described Security Patterns

We also observed that some of these patterns had been described more than once. In total, 77 (18.6%) duplicates were identified by the use of similar names and then comparing their descriptions sketchily. The multiply-described patterns are highlighted in Figure 3.5 and are also given in Appendix B. The most frequently described pattern is the Authenticator (five times). The Controlled-Object Factory, Controlled-Process Creator, and Controlled-Object Monitor pattern are described four times, followed by three descriptions for the following patterns: Multilevel Security, Role-Based Access Control (RBAC), Secure Logger, Authorization, XML Firewall, Fail Securely, Single Access Point, Application Firewall, Execution Domain,
Checkpointed System, File Authorization, and Session. The other 61 patterns are only described twice in the inspected period of time (1997 to 2016).

Hence, they were filtered out, and the number of patterns reduced to 430. The use of similar names identified these duplicates. Afterward, their descriptions were compared sketchily. Due to the abundance of patterns and the non-uniform descriptions of the patterns, we did not check in depth whether two patterns with different names relate to the same concept. The issue of the non-uniform descriptions will be discussed briefly in the Section 4.3 and extensively in Chapter 5.

### 3.2 Summary

In this chapter we have collected several security patterns of different venues. This is a step towards a centralized location for these patterns as suggested by de Muijnck-Hughes and Duncan [103]. Such a single location is beneficial to developers and other users by providing an overview of existing patterns. Moreover, we have made several observations on the security pattern publication landscape:

- Security patterns are mainly published at pattern conferences and rarely at other conferences.
One author is dominating the scientific community publishing security patterns which can decrease the potential of objectivity and self-regulation within the scientific community.

18.6% of the collected security patterns are multiply-described patterns.

These three factors may have contributed to the observed unpopularity of security patterns in the field of software security [238, 447].

Another critical application factor of patterns is selecting a pattern for a given purpose. This process can be supported by a classification. Hence, the next chapter organizes the collected patterns according to two different aspects.
CHAPTER FOUR

CLASSIFICATION

The abundance of documented security patterns in Chapter 3 calls for meaningful classifications to ease searching and assessing the right pattern for a security problem at hand. Moreover, not all security patterns seem to be suitable for a software's implementation as well as the software maintenance and review phase. Thus, the bunch of security patterns must be organized to get a clear picture of what kinds of security patterns exist and to identify software-security patterns suitable to be used for a security pattern recognition approach.

As mentioned before, existing pattern classifications are often based on a small subset of patterns. For instance, Hafiz et al. formed their classification with only 14 security patterns [209]. Their scope is often limited to particular areas such as implementation patterns. Another problem, however, is that information-security experts are rarely development experts [502]. Thus, the usage of existing security patterns and their selection by way of a classification are challenging tasks for non-security professionals who are interested in security aspects. Therefore, two new classification schemes are proposed with the collected security patterns of Chapter 3. The first classification scheme is based on all collected security patterns and shaped towards the selection by application domains, which is relevant for researchers and practitioners who are interested in security patterns. The second shows in detail which security and implementation forces security patterns with respect to software have.

Hence, design and security classifications are inspected to determine their similarities and differences to derive possible criteria for a classification that reflect pattern recognition requirements. The software-security patterns of the application-domain classification are the base for this further distinction. Furthermore, our classification is inspired by pattern-recognition needs and combined with the security issues that these patterns solve. For example, this classification can be used by software developers to choose patterns according to particular security requirements.

This chapter starts with an overview of classifications in general. Existing classification approaches for security patterns are described in Section 4.2 and challenges in categorizing security patterns are depicted in Section 4.3. Afterwards,
Section 4.4 introduces the application-domain classification followed by the specific classification for software-security patterns in Section 4.5. Finally, Section 4.6 discusses the two presented classifications and a brief summary is given in Section 4.7.

### 4.1 Requirements for Classifications

The increasing number of patterns makes it necessary to develop classifications. This section describes requirements for classifications in general.

A classification should be based on systematic methods and techniques to organize a mass of patterns. It organizes patterns into groups of patterns that share one or many properties such as the application domain or a particular purpose. The kind of properties that should be used is not fixed and can be customized according to one's needs. A pattern can have more than one specific property. Therefore, it may be included in more than one classification category.

According to Buschmann et al., a pattern classification scheme should meet some basic properties [72]. It must both be simple and easy to learn. This should be supported by using only a few classification criteria to reduce the complexity for users. Besides, a classification should reflect the main properties of a pattern to classify. Last but not least, a classification scheme should provide the possibility to classify new patterns.

Fernandez et al. pointed out that a classification should make the application of patterns more comfortable along the software life-cycle [158]. Since it is impractical to look at all details of all patterns during pattern selection for the problem at hand, a classification should help to understand the essential nature and value of patterns.

![Intuitive classification](image)

**Figure 4.1 – Intuitive classification.**

A natural way to classify patterns is to categorize them according to the criteria shown in Figure 4.1. A simple and intuitive classification can provide one or more of these criteria:

- **Discipline** - categorize patterns according to the discipline when they are applied such as requirements or reverse engineering.

- **Domain** - differentiate patterns by their application domain such as network, embedded systems, or distributed systems.
Granularity - rank patterns depending on the level at which they address a system, e.g., they may address software design or coding patterns.

Paradigm - sort patterns according to paradigms, e.g., programming paradigms such as object-oriented or imperative programming.

Purpose - order patterns by the kind of problem a pattern solves and the point in time it may be applied.

Scope - organize patterns with regard to the characteristic of using them, e.g., class or object representation (see also [173]).

4.2 Existing Classifications

The presented classification criteria in Section 4.1 are simple, but do not always fit for selecting the right pattern for a special purpose because of their generality. Therefore, more specific classification schemata based on one or more criteria have been developed to meet special purposes. Due to the fact that security patterns are formed according to the archetype of design patterns, this section starts with classifications for design patterns and continues with existing security-pattern classifications. The section is closed by discussing gaps in security-pattern classifications and whether design-pattern classifications can be used to classify security patterns.

4.2.1 Design-Pattern Classifications

The first classification of design patterns (GoF patterns) has been introduced by Gamma et al. [173]. They have classified their patterns based on two criteria: scope and purpose.

<table>
<thead>
<tr>
<th>Scope</th>
<th>Purpose</th>
<th>Creational</th>
<th>Structural</th>
<th>Behavioral</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class Related</td>
<td></td>
<td>Factory</td>
<td>Adapter (class based)</td>
<td>Interpreter</td>
</tr>
<tr>
<td>Object Related</td>
<td></td>
<td>Singleton</td>
<td>Facade</td>
<td>Iterator</td>
</tr>
</tbody>
</table>

Figure 4.2 – GoF classification with a few examples [173].

As depicted in Figure 4.2, the “scope” dimension is distinguished by object composition and class inheritance. The purpose dimension is split into creation, structural and behavioral criteria. A pattern that is related to an object creation fits into the creation criteria. If a pattern is concerned with compositions or structures that are created by classes or objects, it is called structural. The last criterion behavioral deals with the way communication or responsibilities are distributed among classes or objects.
Zimmer organizes the GoF patterns according to their relationships [553]. He classified the relationships in pairs \((X, Y)\) where \(X\) and \(Y\) are different design patterns. The relationships are defined as follows:

- \(X\) uses \(Y\) in its solution,
- \(X\) is similar to \(Y\),
- \(X\) can be combined with \(Y\),

With these categories, he introduces a new layer structure for pattern classification. According to Figure 4.3 relationships and structure of patterns are distinguished into three layers:

- **Basic design patterns and techniques**
- **Design patterns for typical software problems**
- **Design patterns specific to an application domain**

![Figure 4.3 – Zimmer’s classification with a few examples [553].](image)

Later on, Buschmann et al. state that all patterns reside on different abstraction layers and it would be more useful to organize them into criteria that express their abstraction level [72]. Therefore, the authors divide their patterns into three kinds of patterns:

**Architectural patterns** specify the fundamental structure of applications.

**Design patterns** describe often occurring structures of software-component communication that solve a recurring design problem for a specific context.

**Idioms** coding patterns, that is, proven conventions and techniques used during the implementation phase of an application.

Some patterns depend on the technology or domain they are used for and implemented in. These are so-called domain-dependent patterns, e.g., Java Platform, Enterprise Edition (JEE) design patterns. These patterns can be used only in the JEE environment. A system of such patterns has been described by Alur et al. [13].
The authors want to keep the classification simple for their patterns, so they assume “each pattern hovers somewhere between a design pattern and an architectural pattern”. These patterns can be classified in the following categories according to their logical tiers (Figure 4.4). The presentation tier is responsible for creating the presentation used by the client to interact with the user. The business tier is responsible for executing the business logic of the application and applies the business logic to the information received from the integration tier. The integration tier performs the data-access operations for the application.

<table>
<thead>
<tr>
<th>Tier</th>
<th>Patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td>Presentation</td>
<td>Composite View</td>
</tr>
<tr>
<td>Business</td>
<td>Service Locator</td>
</tr>
<tr>
<td>Integration</td>
<td>Service Activator</td>
</tr>
</tbody>
</table>

**Figure 4.4** – JEE pattern classification with a few examples [13].

The discussed design patterns exist since 1994. Classifying them is not a highly active research topic in the design-pattern community. An exception is the emerging of new technologies like JEE which require new classifications or the re-evaluation of existing ones. The older ones were not refined further, except for some theoretical abstractions like the one by Hasso and Carlson [229]. They use a complex algebraic structure to classify design patterns.

In 2006, Shi and Olsson identified a lack of classifications for the need of design-pattern recognition [446]. They suggest a reclassification related to the need of pattern detection by using five categories language provided, structure driven, behavior driven, domain specific and generic concepts. When a pattern is implemented in some programming language and can be identified by looking at the inheritance hierarchy or specific method names, the pattern is part of the language provided category. Patterns that are deeply shaped by their structure and can be identified by their inner-class relationships, such as the Bridge or Composite pattern are structure driven patterns. Patterns that have a structure coupled with a specific behavior fit in the category behavior driven, such as Singleton or State pattern. Patterns, such as Interpreter or Command, serve domain-specific needs. Detecting such patterns requires domain-specific knowledge. They belong to the domain specific category. Patterns in the category generic concepts lack a definite structure and behavioral aspects such as the Memento pattern.

Their classification allows them to exclude the domain-specific patterns and generic concepts, which cannot be found with common behavioral and structural pattern detections. Moreover, they excluded the language-provided patterns from their detection process because of their easy detection using name matching, too. Patterns that reside in the categories behavior driven and structure driven were used for design-pattern detection in their tool PINOT [446]. After Shi and Olsson’s
classification approach, no new design pattern schemata have been developed to the best of our knowledge.

### 4.2.2 Security-Pattern Classifications

One of the simplest classifications for security patterns has been presented by Kienzle et al. [282]. They use the *structural* and *procedural* criteria for the differentiation of the patterns described in their final report. If a security pattern is concerned with compositions or structures that are implemented in a software product, it is *structural*. If a security pattern improves the process for developing secure software with regard to the organization or management, it is called *procedural*.

Konrad et al. [291] have proposed a classification method for security patterns by re-using the classification for design patterns such as *creational*, *structural* and *behavioral* from Gamma et al. [173]. They enhance their classification by adding further categories such as network, host, and application (Figure 4.5). In their work, they consider only the security patterns introduced by Yoder and Barcalow [538].

<table>
<thead>
<tr>
<th>Abstraction Level</th>
<th>Purpose</th>
<th>Creational</th>
<th>Structural</th>
<th>Behavioral</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application</td>
<td>Session</td>
<td>Check Point Authorization</td>
<td>Limited View Full View With Errors</td>
<td></td>
</tr>
<tr>
<td>Host</td>
<td>Session</td>
<td>Check Point Authorization</td>
<td>- - - - - -</td>
<td></td>
</tr>
<tr>
<td>Network</td>
<td>Session</td>
<td>Check Point Authorization</td>
<td>- - - - - -</td>
<td></td>
</tr>
</tbody>
</table>

*Figure 4.5 –* The classification by Konrad et al. with a few examples [291].

Schumacher’s security patterns book offers a new classification system [437]. The classification is based on Zachman’s framework for enterprise architecture [544]. It is presented along two dimensions. One dimension represents different views on the interrogatives “what”, “how”, “where”, “who”, “when”, and “why”. The second dimension shows different information model views such as *business model* or *technology model*. Schumacher et al. enhance this framework by adding the column *security* to emphasize the security view and to be able to address all model levels. They organize only the patterns contained in the book into their classification.

Steel et al. classify their JEE security patterns similar to the Alur et al. approach [13, 464]. They separate their patterns in layers that are typical for the development in the JEE domain such as *Web*, *Business*, and *Web Service*, and added a fourth tier that represents the special issue of *identity management* (Figure 4.6). This classification is designed only for the particular purpose of JEE patterns and does not consider other types of patterns.

A more general approach has been introduced by Rosado et al. [406]. They relate security requirements to security patterns and classify security patterns into two categories: architectural and design patterns.
4.2 – Existing Classifications

Hafiz et al. note that simple security-classification concepts are not sufficient to create a partition of security patterns [209]. Their focus is to classify security patterns by their security impact. Their subset of 14 different security patterns is organized by a classification of application context, a Microsoft classification scheme, the CIA and STRIDE model [472]. The acronym STRIDE contains the concepts Spoofing, Tampering, Repudiation, Information disclosure, Denial of service, and Elevation of privilege. Moreover, they proposed a classification based on a tree structure combined with the STRIDE model to join the software and security view in terms of security patterns [209]. The STRIDE model is typically used for threat modeling including identification and prioritization of security vulnerabilities. It is a common tool for security architects who must prioritize the mitigation effort of security techniques.

A multi-dimensional matrix of concerns that classifies security patterns has been introduced by VanHilst et al. [504, 505]. The idea is that each matrix dimension represents a well-defined list of concerns. To classify security patterns, the primary dimension contains concerns of life-cycle activities, such as domain analysis or requirements engineering. The second dimension differentiates security patterns by their component source type such as new code, legacy, or wizard-code. Other dimensions may hold types of security responses like prevention or mitigation, but they can also be further customized to a user’s need. Their classification has been tested by different members of their team, who added six different security patterns to the classification.

Fernandez et al. state that security patterns are architectural patterns [159]. On that account, their approach deals with two classifications that differ in different viewpoints of security patterns. On the one hand, they introduce a classification by a hierarchy of layers and on the other hand, they propose a classification based on the relationships between patterns by using an automatic relationship extraction and analysis technique. Both classifications are abstract and regard only a small number of security patterns.

Washizaki et al. point out that the previously introduced classifications have only a few dimensions and do not embrace the relations between patterns [516]. Hence, they introduce a meta-model to express the patterns’ properties and relations uniformly. The base is an excerpt of the multidimensional classification dimensions presented by VanHilst et al. [504]. They use the following dimensions for their
approach: Lifecycle stage, Architectural level, Concern, Domain, Type of pattern and Constraint. In addition, they use the three Unified Modeling Language (UML) standard relationship types association, generalization, and aggregation to model relationships between security patterns, for example, the Firewall pattern [437] is the generalization of the Address Filter Firewall [153] and the Application Firewall [107] pattern.

They also propose two instances for the metamodel that represent two points of view, namely pattern-to-pattern relations represented as a pattern graph, and pattern-to-dimension relations modeled as a dimension graph. Their approach has been tested with only eight different security patterns that are close to implementation patterns.

Dougherty et al. [115] organize the 15 patterns of their pattern catalog based on the level of abstraction in software development: architecture, design, and implementation.

Similar to the approach of Hafiz et al. [209], Wiesauer and Sametinger [526] use the STRIDE model and also the Common Attack Pattern Enumeration and Classification (CAPEC) catalog to organize security patterns. Their approach maps security patterns to attack patterns and has been evaluated with 40 security patterns. This classification depicts which security pattern(s) can be used to block loopholes arising from a given attack pattern.

A classification based on undesired properties to prevent flaws in new security patterns and their templates has been developed by Laverdiere et al. [300]. They consider the following properties as undesired for security patterns: over-specified, under-specified, lacking generality, lacking consensus and misrepresented. In combination with the Six-Sigma approach they organized 12 security patterns in their work.

Sarmah et al. [425] combine the Formal Concept Analysis (FCA) and linguistic metaphors to classify security patterns. They use a trust-based security model as the foundation for building security pattern lattice using FCA techniques.

An automatic selection approach based on text processing and learning techniques has been presented by Hasheminejad and Jalili [224]. They automate the identification process by learning classifiers of security patterns and then create suggestions on suitable security patterns for the extracted classifiers. The approach has been tested with the 46 patterns of Schumacher et al.’s book [437].

The needs of software developers can also be used to organize security patterns by considering ontological interfaces [281]. Therefore, Khoury et al. differentiate software developer profiles based on designing or coding skills and map them to security terms such as threat models, security bugs or security errors.

Yoshioka et al. propose a classification of security patterns based on software lifecycle phases [539]. For their two-dimensional approach the requirements and analysis phase, architecture and design and, implementation phases and the security concepts such as risk, threat, and countermeasure are used. The goal of this classification is to identify in which software development phases more security patterns are needed.
4.2 – Existing Classifications

Alvi and Zulkernine’s classification is based on the natural cause of security breaching in software applications [14]. Their classification uses also the software lifecycle phases to organize software security patterns. Moreover, they map them with the Common Weakness Enumeration (CWE), sources of flaw taxonomy and attack patterns of the CAPEC catalog. This approach aims to provide easy use of software security patterns to prevent the causes of security violations.

4.2.3 Classification Similarity

An evolution of design and security pattern classifications exists with respect to the used classification ideas. The influence among security and design-pattern classifications is shown in Figure 4.7. Some ideas like the purpose of the GoF’s design-pattern classification has been reused by Konrad et al.’s security-pattern classification. Moreover, the criterion structural has been adapted by Kienzle et al., whereas the criterion procedural and behavioral in the GoF classification have different meanings. Procedural is used with respect to process patterns for the management or organization of software development in contrast to behavioral from the GoF classification where patterns are only software patterns that are implemented in a software system.

The criteria architectural and design of Buschmann’s classification scheme [72] are picked up by Rosado et al. [406] and used in conjunction with requirements for a new classification schema. Analogical, Sarmah et al. [425] adopt the usage of complex algebraic structure that has been previously proposed by Hasso and Carlson [229] to classify security patterns.

The three-tier JEE classification [13] and the four-tier classification for security patterns [464] are very similar, too. They differ only in one additional tier by the security patterns, which deals with identity information. The other three tiers have different names Presentation/Web, Business/Business and Integration/Web Service, but describe the same criterion (see Figure 4.4 and Figure 4.6).

Organizing patterns according to their relationships has been introduced by Zimmer and reused by Fernandez et al. and Washizaki et al., but their relations are different. Zimmer depicts a graph with only three predefined types of relationships and Washizaki et al. focus on the UML standard to represent pattern relations like generalization or use relationships [516, 553]. In contrast to that approach, Fernandez et al. use automatically extracted relationships based on the pattern description [159].

4.2.4 Classification Distinction

We have seen that design-pattern classifications influenced some security-pattern classifications. All published classifications have one element in common: they take only a small number of patterns into account. On the security-pattern side, the used patterns are often very similar to the patterns first introduced by Yoder and Barcalow [538] and on the design-pattern side the approaches often consider the “core” design patterns described by Gamma et al. [173]. Both subsets of patterns
are patterns that are implemented in a software system. This may lead to the impression that only a handful security and design pattern exist and imply that only programming issues are covered by these patterns. However, Henninger and Corrêa [236] showed in 2007 that there are more design patterns than the few patterns published by Gamma et al. [173] and Buschmann et al. [72]. This statement can also be extended to security patterns, which can also describe enterprise or other security-related issues.

A security view is often added to these classifications by using common-threat modeling such as STRIDE [472]. Adding new views increases the complexity of a classification. A problem, however, is that information security experts are rarely development experts [502]. Moreover, increasing the complexity in security-pattern classifications can make the usage of a classification more difficult for users that have no knowledge or experience with security. It may also lead to difficulties in understanding for other interest groups within the addressed security-pattern audience.

Most security-pattern classifications are quite complex by covering more properties or splitting purposes or domains into more dimensions than design-pattern classifications. Therefore, it can be assumed that the security pattern community is aware of the security pattern’s heterogeneity, which reflects the additional dimensions in the security-pattern classifications. Design-pattern classifications are often tailored to one group of interest—the software developers. Hence, they are focused on helping to choose the right pattern in design and developing to achieve
a good code quality and system structure. Due to the fact that the security-pattern audience has more than one group of interest, such as security, software development, or enterprise-process design, the security-pattern classifications are built from more heterogeneous criteria than the design-pattern classifications. Yet, not all interests can be covered by one classification. Therefore, more security classifications have been developed until now and developing new ones is still a current topic in the security-pattern community.

4.2.5 Resumé

Security patterns related to software can be categorized in a way similar to design patterns. Security patterns that describe other aspects than software-related issues cannot be distinguished by the criteria the aforementioned design-pattern classifications offer, such as "tier", "class related" or "language provided". Therefore, the classification process for security patterns is split into two steps. First, all security patterns are organized according to their application domain with respect to their heterogeneity. Second, the focus is on software-related patterns and a new classification with existing criteria of design-pattern classifications with respect to software-security patterns is developed.

4.3 Challenges in Categorizing Security Patterns

Identifying duplicates is not the only challenge in categorizing security patterns. Yoshioka et al. and Heyman et al. assert that the abstract description of patterns is another challenge [238, 539]. Since the quality of the security-patterns descriptions may influence the categorizing outcome, some time is spent on inspecting the description forms.

4.3.1 Description Form Inspection

Various descriptive models for security patterns exist within the security-pattern community [436]. The POSA\textsuperscript{1} model described by Buschmann et al. is said to be frequently used to describe the context and usage of security patterns [72]. During the priorly conducted literature review (Chapter 3), we observed that the patterns are often described in custom styles and do not follow strictly the POSA model.

We assume that description sections, which are often used in several pattern descriptions, provide the best clues for selecting and organizing security patterns. Therefore, all used description sections in the collected security-pattern publications are gathered to review how security patterns are described formally.

\textsuperscript{1}POSA is the acronym of the design-pattern book series “Pattern-Oriented Software Architecture” written by Buschmann et al. [72].
### 4.3.2 Section Assessment/Examination

There are 70 different sections or aspects, such as *Problem*, *Intent* or *Known Use*, that are used in the 91 collected security pattern publications. The heterogeneity in naming the sections is very high and no mapping between the different names exists like the one by Henninger et al. between the POSA and GoF descriptions [236]. For this reason, significant sections are identified. They can be used to obtain a first impression on security patterns and make patterns comparable (Table 4.1). An overview of all collected sections is given in Appendix A.

**Context** is a frequently used description section with 49 of 91 hits, but the context description is often quite short. In some publications, it only consists of one or two sentences (e.g., [59] or [156]). This circumstance makes it hard to obtain sufficient knowledge or even an idea of what the pattern is about. Laverdiere et al. have made similar findings for the naming and the section *Intent* in security-pattern descriptions [300].

The *Problem* section occurs in about 84 percent of the publications. In many cases, these problem descriptions are abstract or describe a simplified problem for the security pattern (e.g., [59]). Therefore, this description section is less applicable to categorizing security patterns for an application domain but suitable to gather the security aspects it addresses.

The *Related Patterns* section requires a profound knowledge of other security patterns and their application domains to be used as a distinction.

This also applies to the *Consequences* section where additional knowledge is required to be able to relate to application consequences in the security area. Hence, these sections cannot be recommended for novices in security to accomplish a pattern distinction.

A *Known Use* section depicts where a pattern can be found in real life. This section often labels software or software parts like an application login screen, UNIX telnet or Linux as operating system software. The given keywords and explanations within this section give a good impression to which domain this pattern can be applied.

A *Solution* section is used in about 87 percent of the 91 publications. The described solution in the collected publications is frequently used and often provides a good depiction of the security pattern and the problem it solves. Moreover, this

<table>
<thead>
<tr>
<th>Used by Publications</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>79</td>
<td>Solution</td>
</tr>
<tr>
<td>76</td>
<td>Problem</td>
</tr>
<tr>
<td>72</td>
<td>Consequences</td>
</tr>
<tr>
<td>67</td>
<td>Context</td>
</tr>
<tr>
<td>65</td>
<td>Known Uses</td>
</tr>
<tr>
<td>62</td>
<td>Related Patterns</td>
</tr>
</tbody>
</table>

*Table 4.1 – Sections which are often used by security-pattern description forms*
4.4 – Challenges in Categorizing Security Patterns

section often describes which security aspects are covered by the pattern and how this could be implemented by software-security patterns. If the description does not provide a solution section, it can hardly be considered a pattern description. One may argue that the GoF description template neither provide a solution section, but they describe this aspect in a refined manner using the description sections Participants, Collaborations and Implementation. This heterogeneity in the form of descriptions is one aspect that has to be considered and will be discussed in the following section.

4.3.3 Challenges for the Classification Approach

Besides the high variation in the pattern description quality, not all often occurring aspects are equally useful to obtain a quick access to a pattern’s goal and application domain.

If one does not have the time to read a whole pattern description or has a lack of sufficient security knowledge to understand the described pattern, one should look at first at the Known Use aspect to get an idea of the pattern’s application domain. On account of the good depiction of the security pattern and the problem it solves in the Solution aspect, it can be recommended in a second step to look at this section, if no Known Use section exists or if the containing information is not satisfactory. In addition, the Solution aspect gives hints on how the pattern can be implemented in software or used for end users or enterprise processes.

With this strategy, approximately 80 percent of the patterns can be sufficiently understood. The remaining 20 percent can only be organized by reading the full pattern description because of their insufficient description structure in comparison to the majority of security publications. Thus, due to the high description heterogeneity and high varying description quality, the whole pattern description was read for each classification.

During the pattern description-form examination process we observed that some description-form aspects are filled in an insufficient way. An example is the publication by Yskout et al. where many aspects in the pattern description form exist, but many of them are filled with one or two words or with a few sentences [540]. Due to the fact that such an imprecise description leaves much room for interpretation and imagination about what the pattern describes, it increases the difficulty in the distinction process for a new classification. It may also compromise the correctness of distinction.

Many security-pattern description forms follow in some aspects the POSA template, but they are compounded by different terminology like Problem and Motivation or See Also and Related Patterns, which describe the same issue in the publications. A uniform form of description is desirable. Research should aim at improving the quality of security-pattern descriptions. Initial work along this line has already been done but only for a small subset of patterns [213, 238].

41
4.4 Application-Domain Classification

The new classification unifies the existing patterns into a common scheme. In addition, not every task needs information about attack surfaces or vulnerability classification properties like STRIDE or other facets that are introduced in Section 4.2. On that account, specialized criteria like STRIDE are omitted and the focus lies on universal differences among the security patterns. With this in mind, a new classification with a more general perspective based on a domain criterion (Section 4.1) and the security patterns collected in the systematic literature review (Chapter 3) is developed.

4.4.1 Organizing by Application-Domain

To derive the classification, the data and collected keywords were skimmed for the security patterns, such as "user", "password", "operating system", "enterprise" or "process". These keywords were gathered by the information found in the pattern descriptions.

![Figure 4.8 – Proceeding steps in our classification model.](image)

In the next iteration, keywords were extracted by going through the pattern list. On further reading, these keywords were unified into common groups. For instance, the keywords AIX, Linux and Preforking were unified into the group Operating System. The result contains a mixture of purpose and domain criterion. 13 different groups were formed this way. To further simplify the classification along the lines described in Section 4.1, these keywords were further condensed to form an application-domain based distinction, which is easy to understand and intuitively applicable (Figure 4.8). Finally, Figure 4.9 depicts the five target application domains that were discovered: Enterprise, Software, Cryptographic, User, and Network. They are described in the following in more detail.

4.4.2 The Application-Domain Criteria

**Enterprise**-security patterns deal with aspects that are important for enterprises to ensure security in several enterprise segments such as third-party communication
4.4 – Application-Domain Classification

Figure 4.9 – Application-domain based classification.

with suppliers. This means security in processes, physical authentication to several areas, risk mining or securing communication in internal and external businesses. A good example of this pattern type is the Manage Risk pattern introduced by Elsinga and Hofman [131]. The problem addressed by this pattern is as follows: “What is the right (combination of) paradigm(s) to formulate the corporate security strategy in order to select and implement the appropriate set of security safeguards?” [131]. The pattern suggests instructing people and units to pay attention to known and unknown risks to develop prevention and roll-back strategies.

Network-security patterns address network infrastructures and their ideal composition. For instance, the Packet Filter Firewall pattern [437] describes how to shield an internal network from Internet attacks just by tunneling the communication traffic through a single controllable instance and the Virtual Private Network pattern [436] depicts how secure connections over public networks such as the Internet can be established. The Point-to-Point Tunneling Protocol (PPTP) is a specific implementation of this pattern [484].

User-security patterns are focused on user behavior or her awareness of security issues, for example, the Password Lock Box pattern, which encourages the user to protect master passwords with the highest level of security [391]. It stresses the significance of protecting master password files and depicts situations where such a file can be useful. The Keep It Secret pattern [391] highlights that published or publicly known passwords pose a potential danger to be misused by attackers. To minimize this effect, one should keep a password secret or use Password Salt (another security pattern) to vary the password [391]. Another pattern in this domain describes how one can configure the web browser to control how and when cookies are set and used [433].

Software-security patterns describe mostly how to structure parts of a software to ensure security requirements. Sometimes they also describe a specific behavior or way to manage or control the data flow securely. On the one hand, patterns in this domain can be particular like JEE patterns, which can be applied only to Java enterprise applications [464]. An example is the Container Managed Security pattern [464], which is a standard way to enforce authentication and authorization in a JEE application so that no particular hard-coded security policies are necessary. On the other hand, patterns in this domain can be more general like the Single Access Point pattern, which models a kind of login structure that can be found in several software systems like UNIX, ICQ or Twitter [538]. Patterns of this application domain can
also be called *Security Design Patterns* along the lines of the GoF design patterns, which also focus on software.

**Cryptographic** security patterns depict secure communication between two applications over a network. They are often described abstractly. Therefore, it is not clear whether these patterns reside in the *Network* or *Software* domain. Their implementation or application is possible in both domains. On that account, they are seen as a part of network and software in our classification (Figure 4.9). An example is the *Sender Authentication* pattern. It presents the problem and solution how to guarantee that a received message has been sent by a person one expected [59]. Such a pattern can be applied at the network level (level 3 and 4) or the application level and depending on that, it resides in the *Network* or *Software* application domain.

### 4.4.3 Result

The aforementioned classifications in Section 4.2 cover only parts of the discovered domains. The *Network* domain is partly touched by the classification of Konrad et al. [291]. Schumacher et al. factor *Enterprise* requirements customizable with viewpoints in their classification, but they do not distinguish other domains as the presented approach does [437]. The domains *User* and *Cryptographic* are not mentioned in the existing classification approaches, although they represent approximately 16% of the patterns (Figure 4.10). All organized patterns with their application domain(s) are listed in Appendix B.

![Pattern distribution per domain](image)

*Figure 4.10 – Pattern distribution per domain.*

### 4.5 Merging Pattern Recognition and Security Needs

The application-domain classification scheme can be tailored further to practical or research interests by employing, for example, viewpoints as recommended by
Fernandez et al. [159]. For software engineering, in particular, applicable patterns are located in the category *Software*, which can be further divided into specific purposes such as pattern detection by using the existing pattern classifications by Shi and Olsson [446]. Developing new viewpoints or finer grained classifications to cover new needs in terms of special purposes for one of the application domains is also conceivable.

An interesting issue for us is the effect of implemented security aspects in software. The usage of security patterns can harden and protect a software application against common threats [207, 212]. Furthermore, architectural risks for a software system could be determined through the fact of usage or non-usage of security patterns [215]. Therefore, assuming that software which is designed using security patterns is more secure, and it is possible to detect these patterns, a reviewer or maintenance programmer can rate the degree of security of a software system or parts of it by the usage of security patterns. This is comparable to software design patterns, which imply a higher code quality when used appropriately.

### 4.5.1 Classification

This approach is motivated by searching for security patterns in software to be able to determine the built-in security mechanisms of a software system. The application-domain classification scheme indicates which security patterns are relevant for designing software. Due to the fact that this classification scheme is very general, the patterns in the *Software* domain are tailored further to the research goal of this thesis. To show the pattern’s security impact and its purpose in terms of software development two dimensions were chosen for this classification. The first dimension represents the pattern-recognition aspects and the second common security aspects (Figure 4.11). All classified software-security patterns are given in Appendix C in detail. The classification dimensions are described in depth in the following sections.

### 4.5.2 Pattern-Recognition Aspects

Early approaches to design-pattern detection date back to the year 1996 [293]. Several specialized approaches to pattern recognition exist that use different aspects of patterns for their detection, such as structural, behavioral aspects or software metrics. The common matching techniques are all based on structural and/or behavioral aspects, e.g., the approaches by Keller et al. [278], Kramer and Prechelt [293], Shi and Olsson [446], and Wendehals [524]. Patterns of these aspects need different information and analyses to be automatically detected in a software system. Having a distinction for these aspects is very helpful to define which analyses and what kinds of information are needed to find a specific pattern.

Some existing security-pattern classifications depicted in Section 4.2 are formed on ideas from formerly published design-pattern classifications. Thus, they imply that security patterns are like design patterns. We have seen another picture of the security-pattern landscape with the priorly presented application-domain classification in Section 4.4. Due to the fact that security patterns with regard to
software were extracted from the whole set of patterns, some of the design-pattern classification criteria can be used here to organize them with the pattern-detection application approach of this thesis in mind.

Representing information relevant to pattern recognition, some criteria of the design-pattern classification of Shi and Olsson were chosen [446]. **Structural** software-security patterns are characterized by their particular class structure. This structure can be realized by inter-class relationships like inheritance, association or delegation relationships (see also [446]). The **Single Access Point** pattern is such a structure-driven pattern [437, 538]. It provides single access to a protected system. The focus is on the pattern's structure in a software described through static relations rather than its behavior at runtime. **Behavioral** software-security patterns are primarily designed from a behavioral point of view. They can be easier described and found by their typical behavior than their structure (see also [446]). An example of such a pattern is the **Secure Logger** [464]. It is a class that manages the logging of data in a secure and centralized manner. **Generic Concept** security patterns describe very general solutions for security problems. Unfortunately, many of the patterns in this category do not provide implementation details, UML diagrams or other information, such as, example code snippets, which can be used to distinguish between a structural or behavioral character of the pattern. An example is the **Password Authentication** pattern [282], which describes the secure management of passwords while designing a user login.

<table>
<thead>
<tr>
<th>Recognition Security</th>
<th>Structural</th>
<th>Behavioral</th>
<th>Generic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accountability</td>
<td></td>
<td>Audit Interceptor Secure Logger</td>
<td>Password Authentication</td>
</tr>
<tr>
<td>Authentication</td>
<td>Single Access Point Subject Description</td>
<td>Secure Visitor</td>
<td>Password Authentication</td>
</tr>
<tr>
<td>Access Control</td>
<td>Check Point Subject Description</td>
<td>Secure Visitor</td>
<td>Keep Session Data in Client</td>
</tr>
<tr>
<td>Availability</td>
<td>Partitioned Application</td>
<td>Secure Preforking</td>
<td>Keep Session Data in Client</td>
</tr>
<tr>
<td>Confidentiality</td>
<td>Partitioned Application</td>
<td>Secure Visitor</td>
<td>Password Authentication</td>
</tr>
<tr>
<td>Integrity</td>
<td>Check Point</td>
<td>Secure Preforking Secure Visitor</td>
<td>Password Authentication</td>
</tr>
<tr>
<td>Non-Repudiation</td>
<td>Subject Description</td>
<td>Audit Interceptor Secure Logger</td>
<td></td>
</tr>
</tbody>
</table>

![Figure 4.11](image)

Shi and Olsson also provide the criteria **Domain Specific** and **Language/Framework provided** but they cannot be directly used for our security-pattern classification [446]. Since security patterns are mostly written abstractly, it is not possible to classify a pattern according to these two criteria. A similar analysis must be made for the language-provided patterns which manifest through a framework-specific structure or method names for this type of patterns.
4.5 – Merging Pattern Recognition and Security Needs

4.5.3 Security Aspects

Developers tend to view IT security in terms of software requirements rather than taking the perspective of an attacker. Software patterns are usually chosen by developers with a particular goal in mind. For this reason, general security goals are employed for our pattern classification and not the STRIDE model, which focuses on the attacker’s perspective. In the area of IT security, the most common goals are confidentiality, integrity, and availability of data [21]. **Confidentiality** guarantees the secrecy of data, whereas **integrity** makes sure that data are not modified in an unauthorized way. The **Secure Visitor** pattern fulfills the latter aspect. Nodes can only be accessed by a **Secure Visitor** who prevents unwanted access and unauthorized modifications of nodes in hierarchically structured data. **Availability** means that data/services are accessible. The **Keep Session Data in Client** pattern provides the accessibility to a website if the connection between client and server is interrupted for a short time [475]. Sometimes, **non-repudiation** (proving an action to a neutral and trustworthy third party) and **accountability** (logging certain actions for audits) are also of interest. One pattern example of these two aspects is the **Audit Interceptor** pattern [464]. It intercepts audit requests and responses to and from the business tier in JEE applications and properly logs them. In addition, the identification of principals (e.g., users, machines, and processes), called "**authentication**", is important as well as **access control**, which determines which principal may access which data. Both aspects are used in the **Secure Visitor** pattern where the visitor has to verify a user’s credentials and check it against the access control rules for modification [437]. As a consequence, our classification considers confidentiality, integrity, availability, non-repudiation, accounting, authentication, and access control in the second dimension.

4.5.4 Result

All software-security patterns described in Section 4.2 were organized according to the aforementioned aspects (see Appendix C for details). The software-security patterns often describe integrity and access control problems (99 and 71 times, respectively; see Figure 4.12). Confidentiality and authentication issues are often used, too. In total, 69 and 62 patterns can be found per criterion. Lesser attention in the software-security patterns have the security aspects availability, accountability, and non-repudiation with 28, 12, and 4 patterns, respectively, which deal with these problems. As depicted in Figure 4.12, no structural-driven pattern was found that covers accountability and no generic-concept pattern that handles non-repudiation aspects. All other security criteria were matched by a software-security pattern.

In total, 73 **behavioral** and 43 **structural** characterized software-security patterns were detected. The majority of the patterns belong to the **generic concepts** criterion. **Generic concept** patterns cannot be directly used for a pattern recognition approach. Most of these patterns do not provide implementation information like example code that could enable a distinction in structural or behavioral. A distinction can be made if further inspections on the real usage of these patterns in software systems have
been conducted and concrete implementations can be found and assessed. With this additional investigation, it may be possible to detect such patterns in the future and give designers a better idea of how to design and implement these patterns in a software system. Some of the generic concept patterns like Red Team The Design [282] are expected to have no implementation in software.

### 4.6 Discussion

The presented application-domain classification scheme fulfills the requirements of classifications in terms of expandability, intuitive use, and is applicable for security laymen. This approach can be expanded by repeating the proceeding steps described in Section 4.4.1 for new patterns if new application domains for security patterns emerge. The intuitive use and the applicability for non-security specialists is supported by the use of only one criterion—selection by domain—which is easy to decide for a user. Supposing that a user knows in what domain she will work with security patterns, e.g., an enterprise process designer may select Enterprise as her application domain, the security pattern set of 130 decreases to 94 enterprise-security patterns.

The application-domain classification is expected to help other researchers and practitioners with specific application goals focusing on security patterns. A possible use case for this classification is an enterprise process architect looking for a best practice to administer threats and risks for her enterprise. Then she can have a look at all enterprise-related patterns listed in Appendix B and will find patterns like Risk Determination and Threat Assessment to solve her problems [437].
Furthermore, the second classification can help software designers to choose the right security patterns for their software system according to specific security requirements. This classification also gives a detailed overview of which patterns can be used in general for software-related issues. Moreover, it allows one to select a software-security pattern according to its security attributes for the software design phase or determining security features for a security assessment.

Besides the open issue—which concrete similarities and differences design and security pattern have [503]—two additional gaps in research have been identified. One is that additional work must be done to define a uniform description for security patterns to increase the description quality. As mentioned in Section 4.3, some work has been done in this direction, but we have also seen that the heterogeneity even by newer pattern publications is still very high. On account of this, it is desirable to have all security patterns available from a single source and presented in a uniform format like other existing open databases for design patterns (e.g., [331, 535]).

Another open issue is that no structural patterns with accountability aspects and no general concept patterns with non-repudiation properties were found. The absence of these aspects indicates a gap in the software-security pattern landscape.

All in all, additional investigations are necessary for all software-security patterns not only generic concept patterns. For the second classification, it is easy to decide whether a pattern falls into the category of structural or behavioral patterns based on their descriptions, but they are often not sufficient and exact enough to use existing pattern recognition approaches out of the box for their detection. It remains a high variability in their possible implementations.

4.7 Summary

This chapter has shown a comparison of design and security-pattern classifications, discussed challenges in classifying security patterns, and introduced two new classification schemes.

The first classification scheme embraces 430 published security patterns and exceeds in numbers existing classifications by far. The second classification combines the focus of pattern recognition and security aspects. It classifies more than 190 software-security patterns which have been obtained in the first organization process. The two classifications are used to distinct security patterns with regard to the software domain and further specializes the software-security patterns to pattern detection criteria. We use the organized software-security patterns as basis of the next steps in this thesis to validate and detect software-security patterns implemented in code.
Patterns, in general, are presented with a special description form that depicts the pattern’s specific properties. As we have seen in Section 4.3.1, such a description form consists of different sections, for instance Motivation, Problem or Solution. Section 4.3 has already marked out that the description forms of security patterns are very different. This heterogeneity is a problem for comparing security patterns, their comprehension, and application in the software design, and usage of tools that should support their selection in the software development process. Thus, this chapter examines the security-pattern descriptions of the software-related patterns identified in Chapter 4.

5.1 Introduction

Many patterns have been published since the first publication of security patterns in 1997 [538]. In Section 3.1.2, 91 publications for the period of 1997 to 2016 that describe security patterns have been collected. In total, these publications contain over 528 patterns whereof 267 patterns can be classified as software-security patterns and 72 of them are multiply described (see Chapter 4). Besides the common design pattern templates (POSA and GoF) various descriptive models for security patterns exist within the security-pattern community ([436, 539] and Section 4.3.1). For example, this is a challenge when a unique template is desired to provide tool support [320] or, in our case, multiply described patterns shall be analyzed and the fusion of their descriptions shall be used to shape a better-documented pattern.

This variety can be useful to stress pattern-specific properties. Besides the sections used in POSA and GoF templates, the sections that have additionally been introduced in the published software-security pattern descriptions possibly fulfill this requirement. They may be used for describing new security patterns in future to stress their security potential. These newly introduced sections raise the following question:

**Question 1** – Do security-specific description sections in published software-security patterns exist?
The heterogeneity of sections also incurs some disadvantages. It can hamper the comprehension and application of security patterns, and it is often difficult to notice that two different descriptions relate to the same pattern if described in non-uniform ways (Section 3.1.2). In addition, the heterogeneously described patterns must be transformed into one tool template if tool support for selecting security patterns is desired; e.g., María et al. proposed such a tool [320]. To decrease the heterogeneity, some researchers like Laverdiere et al. proposed new description templates with new sets of sections [300]. Despite these efforts, the description heterogeneity does not decrease (Section 4.3.2). This may indicate that the additionally proposed sections are not essential or do not provide an improvement over describing the security properties of a pattern with regard to commonly used sections. This leads us to the following question:

**Question 2** – Can existing description sections be mapped to the intersection of common POSA and GoF description forms?

The description sections used in published software-security patterns are analyzed to address the questions 1 and 2. First of all, the description forms of published software-security patterns (including the multiple described patterns) are compared with common POSA and GoF pattern description forms. This comparison shows which sections have been added and if every description form uses additional sections to describe software-security patterns. The collected additionally used sections are the base for addressing Question 1 by analyzing the definition and usage to find out if these sections are security related. To address Question 2, all security pattern templates are checked if they use additional sections and if these sections can be aligned with common GoF and POSA sections which may indicate that they are not essential for representing a software-security pattern.

The developed alignment of existing sections can reduce the effort of developing a single core format for security patterns, e.g., for building a security-pattern selection tool. It also provides comparability among published patterns. The feasibility of the developed alignment on comparing software-security patterns is tested in a case study on collected multiply-described software-security patterns. To provide the sustainability of this approach, it aims to encourage the community to achieve an agreement on a core set of sections to describe software-security patterns. Such an agreement can decrease the heterogeneity of description forms and makes the patterns comparable. Moreover, through a base set of sections, pattern duplicates can be identified more efficiently, a good description quality can be ensured and be a basis for tool support in the future.

The remainder of this chapter is structured as follows. Two common pattern templates with the description forms of published software-security patterns are compared in Section 5.2. This is followed by the alignment of description sections and security-specific sections in Section 5.3. In Section 5.4 the developed alignment will be used to compare multiply described software-security patterns. This chapter closes with a summary in Section 5.5.
5.2 Identification of Additionally Used Sections

A good pattern description should give one the possibility to determine whether a pattern applies to a particular situation and how the actual instantiation of the pattern should be done. Fernandez et al. stated a sufficient description as follows: “A description model has to provide enough detail and guidance for developers to incorporate them in their applications.” [158]. Alexander et al. suggested that “Each pattern is a three-part rule, which expresses a relation between a certain context, a problem, and a solution.” [11]. Not all pattern-description forms, however, are based on this simple three-part rule. Gamma et al. introduced their model for design patterns in 1994 [173]. Two years later, Buschmann et al. introduced a different model for describing design patterns [72]. These two models formed the basis for other pattern description forms [236, 437].

Within the security pattern community, various descriptive models for security patterns exist ([432, 539] and Section 4.3.2). Therefore, the description forms used in software-security pattern publications were compared to GoF and POSA to determine which sections have been added. These additionally used sections may
indicate that they are necessary to describe the security aspect of a pattern. This issue is checked subsequently.

<table>
<thead>
<tr>
<th></th>
<th>Blakley et al. [50]</th>
<th>Fernandez and Sini-baldi [150]</th>
<th>Steel et al. [464]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Related Patterns</td>
<td>Related Patterns</td>
<td>Related Patterns</td>
<td></td>
</tr>
<tr>
<td>Intent</td>
<td>Intent</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Applicability</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Collaborations</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Motivation</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Participants</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>only GoF</td>
<td>Problem</td>
<td></td>
<td>Problem</td>
</tr>
<tr>
<td></td>
<td>Solution</td>
<td></td>
<td>Solution</td>
</tr>
<tr>
<td></td>
<td>Context</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Variants</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Dynamics</td>
<td></td>
<td></td>
</tr>
<tr>
<td>only POSA</td>
<td></td>
<td>Problem</td>
<td>Solution</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Name</td>
<td>Name</td>
<td>Name</td>
<td>Name</td>
</tr>
<tr>
<td>Consequences</td>
<td>Consequences</td>
<td></td>
<td>Consequences</td>
</tr>
<tr>
<td>Structure</td>
<td>Structure</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Known Uses</td>
<td>Known Uses</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Also Known As</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Implementation</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>both</td>
<td>Forces</td>
<td>Forces</td>
<td>Forces</td>
</tr>
<tr>
<td></td>
<td>Security Factors and Risks</td>
<td></td>
<td>Security Factors and Risks</td>
</tr>
<tr>
<td></td>
<td>Reality Checks</td>
<td></td>
<td>Reality Checks</td>
</tr>
<tr>
<td></td>
<td>Strategies</td>
<td></td>
<td>Strategies</td>
</tr>
</tbody>
</table>
|                 | Participants and Responsibilities |                                          | Participants and Responsibili
ties |
| other           | Forces                        | Forces                                       | Forces                     |
|                 | Security Factors and Risks    |                                              | Security Factors and Risks |
|                 | Reality Checks                |                                              | Reality Checks             |
|                 | Strategies                    |                                              | Strategies                 |
|                 | Participants and Responsibilities |                                         | Participants and Responsibilitie
ts |
|                | GoF-like                      | POSA-like                                    | Not Decidable              |
|                |                               |                                              |                            |

Table 5.1 – An excerpt of description sections of three security pattern publications [50, 150, 464], their similarity markings (only GoF, only POSA, both, and others), and rating (highlighted in italics).

### 5.2.1 Analysis

A security pattern publication can contain more than one security pattern. These publications describe the containing security patterns always in the same way. Based on the conducted literature review in Chapter 3 and the selected patterns of the Software domain in Chapter 4, we identified 48 publications that describe software-security patterns. Based on them all different sections that are used for the pattern description are collected (Figure 5.1). Detailed information on the collected section definitions are listed in Appendix A. For every publication, the used sections are listed. Moreover, the sections that can be found only in POSA or GoF or both description structures are marked. An excerpt of this analysis is shown in Table 5.1.
5.2.2 Result

There are 62 different sections which are used in software-security pattern publications, such as Motivation, Solution or Forces. The majority of publications (36 of 48) use a mixture of GoF and POSA sections (Figure 5.2). Seven publications use solely a subset of POSA [144, 148, 208, 357, 360, 437, 522] and another five publications use GoF sections exclusively in their description template [50, 115, 201, 203, 270]. Among these publications none use any additional section to describe software-security patterns. An additional observation was that the number of used sections varied. The minimum of used sections is three [208], the maximum is 15 [552] and the average is approximately ten sections.

![Figure 5.2 – Overview of POSA, GoF, and additional section usage.](image)

Most of the used pattern templates can be assigned to the GoF or POSA description form. 39 of 48 security-pattern descriptions use a more POSA-like form and only six publications use a GoF-like structure. Due to the similar amount of GoF and POSA sections it cannot be decided on three publications whether the description form is more similar to GoF or POSA (Figure 5.3). Based on the quantity, the POSA template is favored by authors to present software-security patterns.

All in all, design and software-security patterns share basic sections, and it is possible to describe security patterns without using additional description sections. However, 23 publications (approx. 47%) use additional sections and in total 41 different sections can be counted.

The detected additionally used sections are the basis for the mapping in the next section. Furthermore, they are used in Section 5.3.3 to discuss whether they are security-specific.

5.3 Description-Form Alignment

The mixture of common GoF and POSA description-form sections with new ones leads us to the question of whether these “new” sections only have different names for the same content or truly add new sections to the pattern description. Most columns of the GoF and POSA description form can be aligned based on their meaning [236]. Some sections have the same meaning, but different terms are used,
such as Also Known As and Alias. Less apparently, other sections are also similar, such as Applicability (GoF) and Context (POSA). Such an alignment allows the pattern reader to compare patterns expressed in different description forms more easily. Table 5.2 depicts the GoF and POSA alignment developed by Henninger and Corrêa [236].

<table>
<thead>
<tr>
<th>GoF [173]</th>
<th>POSA [72]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Name</td>
</tr>
<tr>
<td>Structure</td>
<td>Structure</td>
</tr>
<tr>
<td>Known Use</td>
<td>Known Use</td>
</tr>
<tr>
<td>Consequences</td>
<td>Consequences</td>
</tr>
<tr>
<td>Implementation</td>
<td>Implementation</td>
</tr>
<tr>
<td>Related Patterns</td>
<td>See Also</td>
</tr>
<tr>
<td>Motivation</td>
<td>Problem</td>
</tr>
<tr>
<td>Applicability</td>
<td>Context</td>
</tr>
<tr>
<td>Also Known As</td>
<td>Solution</td>
</tr>
<tr>
<td>Collaborations</td>
<td>Dynamics</td>
</tr>
<tr>
<td>Sample Code</td>
<td>Example Resolved</td>
</tr>
<tr>
<td>Participants</td>
<td>Variants</td>
</tr>
<tr>
<td>Intent</td>
<td>Example</td>
</tr>
<tr>
<td></td>
<td>Summary</td>
</tr>
</tbody>
</table>

Table 5.2 – Mapping between design pattern sections [236].

Authors of security patterns often use pattern description forms that are based on the one defined by Buschmann et al. [72] (see Section 5.2.1). Possibly, the description sections of published software-security patterns can be aligned with each other to make insufficiently written patterns usable or even comparable to others. In addition, a unique pattern description like Pattern Language Markup Language (PLML) can be applied to security patterns or enhanced to security pattern needs [164]. This can be a first step towards an extensive security pattern catalog with unique descriptions.
which the actual state of security pattern catalogs, e.g. [282, 437, 464, 540], cannot cope with.

5.3 – Description-Form Alignment

5.3.1 Process

As the first step towards creating a mapping of the different sections, for each collected section a characterization is collected. Fortunately, some authors provide a characterization for each used section within their publication.

Besides the books of Buschmann et al. [72] and Gamma et al. [173], eight security publications provide definitions for their used sections (in total 114 distinct definitions) [50, 97, 115, 144, 342, 437, 464, 540]. Schumacher et al. [437] reference and provide the exact definitions of Buschmann et al., Blakley et al., and Dougherty et al. provide slightly modified definitions of Gamma et al. [50, 115, 173]. 21 of 62 sections have no specification of the pattern feature they describe. In these cases, a section characterization is created by reading the patterns that use this section and capture the described pattern facet. In the next step, sections that are defined several times by different authors are checked if they do not describe distinct pattern facets by using the same section name. None of these multiply defined sections differ. Thus, one of them is selected to represent the section for the comparison process. All section definitions are listed in Appendix D.

The characteristics of the additionally used sections (see Section 5.2.1) are compared with each other and the specified sections of the POSA and GoF description forms. If a section describes almost the same or a part of another section, they are aligned with each other. An example is shown in Table 5.3.

<table>
<thead>
<tr>
<th>Section</th>
<th>Characteristic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variants</td>
<td>“A brief description of variants or specializations of a pattern.” [72]</td>
</tr>
<tr>
<td>Strategies</td>
<td>“Describes different ways a security pattern may be implemented and deployed.” [464]</td>
</tr>
</tbody>
</table>

Table 5.3 – Two similar sections and their characteristics.

5.3.2 Section Mapping

The alignment result is divided into the following four groups.

Assignable Sections

Like Henninger’s alignment is based on meanings [236], in certain cases sections such as the aforementioned Strategies section can be aligned to a GoF and/or POSA section Variants (Table 5.4) as both sections describe variants of a pattern.

These assignable sections can be divided into two groups: same meaning and specialization of a section, respectively. The example in Table 5.3 shows the same meaning of two sections. Other sections can be aligned with other sections by
describing a part of it in depth—a specialization (in Table 5.4 highlighted with ★). An example of specialization are the sections Alternatives, Benefits, Conflicts, Impairments, Dependencies. They are described and only used by Yskout et al. and—according to their definition—they enhance the Related Patterns section with sophisticated pattern relationships [540]. Thus, these sections model a part of the Related Patterns section. Two other examples are the Forces and Properties sections that are a part of the Problem section and highlight addressed security issues. The Class-Diagram section presents and describes a class diagram of the pattern that can also be found in the Structure section of other publications. Known Uses is specialized by Non-Security Known Uses in presenting a non-security relevant usage of the pattern. Trade-Offs, Liabilities, Pitfalls, Contraindications and Security Factors & Risks highlight parts of the Consequences section. The last section describes the advantages and disadvantages of applying a pattern.

Some sections can be aligned with other sections by describing a part of it in depth. An example is Consequences where the Trade-Offs and Liabilities sections describe a particular part of a pattern’s benefit.

<table>
<thead>
<tr>
<th>POSA</th>
<th>GoF</th>
<th>Security Patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implementation</td>
<td>Implementation</td>
<td>Implementation, Reality Checks, Issues, Implementation Issues, Implementation Discussion★, Discussion★</td>
</tr>
<tr>
<td>Consequences</td>
<td>Consequences</td>
<td>Consequences, Trade-Off★, Liabilities★, Security Factors &amp; Risks★, Pitfalls★, Contraindications★</td>
</tr>
<tr>
<td>Known Uses</td>
<td>Known Uses</td>
<td>Known Uses, Non-Security Known Uses★</td>
</tr>
<tr>
<td>Problem</td>
<td>Motivation</td>
<td>Problem, Motivation, Forces★, Properties★, Security Objectives★</td>
</tr>
<tr>
<td>See Also</td>
<td>Related Patterns</td>
<td>See Also, Related Patterns, Relationships★, Alternatives★, Benefits★, Conflicts★, Impairments★, Dependencies★</td>
</tr>
<tr>
<td>Example Resolved</td>
<td>-</td>
<td>Example Resolved, Rationale, Resulting Context, Resultant Context, Implementation Example, Example Instances</td>
</tr>
<tr>
<td>Variants</td>
<td>-</td>
<td>Variants, Strategies</td>
</tr>
<tr>
<td>-</td>
<td>Also Known As</td>
<td>Also Known As, Alias</td>
</tr>
<tr>
<td>-</td>
<td>Intent</td>
<td>Intent, Abstract, Thumbnail, Summary</td>
</tr>
<tr>
<td>Structure</td>
<td>Structure</td>
<td>Structure, Class-Diagram★</td>
</tr>
<tr>
<td>Example</td>
<td>-</td>
<td>Example, Non-Software Example★</td>
</tr>
<tr>
<td>Context</td>
<td>Applicability</td>
<td>Context, Applicability, Features★</td>
</tr>
</tbody>
</table>

Table 5.4 – Alignment of assignable sections between POSA, GoF, and security patterns. The ★ character highlights a specialization of a section.

Mixture Assignment

A special case is the description section Participants and Responsibilities. This section describes participating objects and classes and their interaction/collaboration
5.3 – Description-Form Alignment

(responsibilities). It unites the GoF sections Collaboration and Participants to one section (Table 5.5). Cuevas et al. [97] adopted this technique and united the POSA sections Problem and Context into one section (Problem/Requirements and Context) that describes the context and problem of a pattern in a single paragraph.

<table>
<thead>
<tr>
<th>POSA</th>
<th>GoF</th>
<th>Security Patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>Collaborations</td>
<td>Collaborations, Participants and Responsibilities</td>
</tr>
<tr>
<td>-</td>
<td>Participants</td>
<td>Participants, Participants and Responsibilities</td>
</tr>
<tr>
<td>Problem</td>
<td>Motivation</td>
<td>Problem/Requirements and Context</td>
</tr>
<tr>
<td>Context</td>
<td>-</td>
<td>Problem/Requirements and Context</td>
</tr>
<tr>
<td>Structure</td>
<td>Structure</td>
<td>Structure &amp; Participants, Pattern Structure</td>
</tr>
<tr>
<td>Dynamics</td>
<td></td>
<td>Pattern Structure</td>
</tr>
</tbody>
</table>

Table 5.5 – Alignment of mixture assignment sections between POSA, GoF, and security patterns.

Natively used GoF/POSA Sections

These sections are used singularly in the security-pattern descriptions and no other section maps to them (Table 5.6).

<table>
<thead>
<tr>
<th>POSA</th>
<th>GoF</th>
<th>Security Patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Name</td>
<td>Name</td>
</tr>
<tr>
<td>-</td>
<td>Sample Code</td>
<td>Sample Code</td>
</tr>
<tr>
<td>Solution</td>
<td>-</td>
<td>Solution</td>
</tr>
</tbody>
</table>

Table 5.6 – Alignment of natively used sections between POSA, GoF, and security patterns.

Non-Assignable Sections

Some sections cannot be aligned with other sections (Table 5.7). These sections are often used in only one publication. The section Classification provides a short classification for the pattern presented by Gondi [180]. The introduced Preconditions section by Cuevas et al. presents all conditions that have to be checked before applying the pattern [97]. Yskout et al. provide the additional section Labels that describes the pattern’s impact on quality requirements, such as performance and usability [540]. An example highlighting a specific application-domain property is the section Social Dependencies which describes the relationships to other agents in a multi-agent environment [342]. Jafari and Rasoolzadegan introduce the Vulnerability Analysis [264]. Since their paper deals with securing four GoF design patterns, the authors discuss the vulnerability analysis of the correlating (GoF) design pattern to this secured pattern within this newly introduced section.
5.3.3 Security-Specific Sections

In total, 23 of 48 publications use sections that differ from the GoF and POSA templates (Section 5.2.1). Thus, it can be assumed that one or more of these new sections can be security specific for describing software-security patterns. Therefore, these additionally used sections are inspected based on their definitions (Section 5.3.1) again to identify security-specific sections. Table 5.8 lists the result of the investigation.

<table>
<thead>
<tr>
<th>Section</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forces</td>
<td>“Describes the motivations and constraints that affect the security problem. Highlights the reasons for choosing the pattern and provides justification.” [464].</td>
</tr>
<tr>
<td>Security Objectives</td>
<td>“The objectives describe the main security objective the pattern tries to solve.” [540].</td>
</tr>
<tr>
<td>Properties</td>
<td>“They describe which security elements the pattern is providing.” [97].</td>
</tr>
<tr>
<td>Trade-Offs</td>
<td>Describes the contributions of a pattern on the following security goals: Availability, Confidentiality, Integrity, Manageability, Usability, Performance, and Cost (used in [282]).</td>
</tr>
</tbody>
</table>

Table 5.8 – Possible security-specific sections.

The sections *Security Objectives*, *Properties* and *Trade-Offs* are only used in one publication, but the *Forces* section is used in 17 publications. Based on the alignment, extracted security-specific sections and the number of usages, the only section that can be seen as security-relevant is *Forces*. Thus, seven of the patterns that use the *Forces* section are reread to check its security relevance. The result is that this fine-grained description of the *Problem* section is often used to stress the security effect of a software-security pattern, but it is not security-specific due to its general characterization.

In general, the identified sections are not strictly security specific. They could also be used in a context different from security. For example, the *Forces* section is suggested for a general description of patterns [330] and used in descriptions of patterns outside the security focus. Finally, the additionally used sections of software-security patterns are not security-specific, but the *Forces* section is commonly used to highlight the security effect of a pattern.
5.4 Case Study

The developed alignment provides the possibility to compare heterogeneously described software-security patterns. Its applicability is tested with a case study on multiply-described security patterns. This case study is organized according to the case study guideline proposed by Kitchenham et al. that includes the following tasks: define the hypothesis and the method of comparison, select the project/object, as well as, analyze and report the results [286].

5.4.1 Context

We have mentioned in Section 3.1.2 that many multiply described patterns exist. They all share the same name, but it is difficult to compare them due to their different description nature. The feasibility of the developed alignment is tested by determining whether they describe the same issue and identify description differences.

5.4.2 Hypothesis and Measurable Terms

According to the described best practice by Kitchenham et al. [286], we define our evaluation hypothesis: Contrasting security patterns using the section alignment is more efficient than not using it. Therefore, the time of contrasting the multiply-described patterns is measured and it has to be decided whether the two patterns describe the same security issue or not.

5.4.3 Object-Selection Process

Due to different security-pattern features, the measured comparison time between different security patterns cannot be used to lead to a meaningful evaluation. One described security issue can be more complex than the other or one pattern is better known by the user than the other. Thus, patterns that are described in more than two different venues (Table 5.11) are selected.

As observed by us and Heyman et al. [238], the description length of security patterns may vary. Thus, the number of words used in the descriptions is calculated to find comparable pattern pairs. Moreover, some patterns provide figures which are sufficient to understand the basics of the pattern. This issue is taken into account with 200 words per used figure.

Excluded Patterns

Most duplicates (in total 27) come from Yskout et al. [540] (Table 5.9 and Table 5.10). They provide a kind of security-pattern catalog where the patterns themselves are not described. They are only cited in the headline but they provide further information about related patterns such as Impairments, Conflicts or Dependencies. Besides, the properties of the Labels section, e.g., Performance or Maintainability,
<table>
<thead>
<tr>
<th>Security Pattern Name</th>
<th>Publication</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1st</td>
</tr>
<tr>
<td>Access Control List</td>
<td>[106]</td>
</tr>
<tr>
<td>Administrator Hierarchy</td>
<td>[154]</td>
</tr>
<tr>
<td>Audit Interceptor</td>
<td>[464]</td>
</tr>
<tr>
<td>Authentication Enforcer</td>
<td>[464]</td>
</tr>
<tr>
<td>Authorization Enforcer</td>
<td>[464]</td>
</tr>
<tr>
<td>Capability</td>
<td>[106]</td>
</tr>
<tr>
<td>Check Point</td>
<td>[538]</td>
</tr>
<tr>
<td>Checkpointed System</td>
<td>[208]</td>
</tr>
<tr>
<td>Container Managed Security</td>
<td>[464]</td>
</tr>
<tr>
<td>Controlled Execution Environment</td>
<td>[142]</td>
</tr>
<tr>
<td>Credential</td>
<td>[341]</td>
</tr>
<tr>
<td>Credential Tokenizer</td>
<td>[464]</td>
</tr>
<tr>
<td>Demilitarized Zone</td>
<td>[437]</td>
</tr>
<tr>
<td>Full View with Errors</td>
<td>[538]</td>
</tr>
<tr>
<td>Input Guard</td>
<td>[424]</td>
</tr>
<tr>
<td>Keep Session Data in Client</td>
<td>[475]</td>
</tr>
<tr>
<td>Limited View</td>
<td>[538]</td>
</tr>
<tr>
<td>Load Balancer</td>
<td>[475]</td>
</tr>
<tr>
<td>Obfuscated Transfer Object</td>
<td>[464]</td>
</tr>
<tr>
<td>Output Guard</td>
<td>[424]</td>
</tr>
<tr>
<td>Policy-Based Access Control</td>
<td>[106]</td>
</tr>
<tr>
<td>Protection Rings</td>
<td>[147]</td>
</tr>
<tr>
<td>Reference Monitor</td>
<td>[142]</td>
</tr>
<tr>
<td>Secure Access Layer</td>
<td>[538]</td>
</tr>
<tr>
<td>Secure Blackboard</td>
<td>[360]</td>
</tr>
<tr>
<td>Secure Broker</td>
<td>[340]</td>
</tr>
<tr>
<td>Secure Logger</td>
<td>[464]</td>
</tr>
<tr>
<td>Secure Message Router</td>
<td>[464]</td>
</tr>
<tr>
<td>Secure Pipe</td>
<td>[464]</td>
</tr>
<tr>
<td>Secure Pipes and Filters</td>
<td>[148]</td>
</tr>
<tr>
<td>Secure Process/Thread</td>
<td>[154]</td>
</tr>
<tr>
<td>Secure Service Facade</td>
<td>[464]</td>
</tr>
<tr>
<td>Secure Session Object</td>
<td>[464]</td>
</tr>
<tr>
<td>Security Association</td>
<td>[50]</td>
</tr>
<tr>
<td>Security Context</td>
<td>[50]</td>
</tr>
<tr>
<td>Security Logger/Auditor</td>
<td>[161]</td>
</tr>
<tr>
<td>Server Sandbox</td>
<td>[282]</td>
</tr>
<tr>
<td>Session</td>
<td>[538]</td>
</tr>
<tr>
<td>Session Failover</td>
<td>[475]</td>
</tr>
<tr>
<td>Session Timeout</td>
<td>[475]</td>
</tr>
<tr>
<td>Session-Based Role-Based Access Control</td>
<td>[149]</td>
</tr>
<tr>
<td>Single Access Point</td>
<td>[538]</td>
</tr>
<tr>
<td>Subject Descriptor</td>
<td>[50]</td>
</tr>
<tr>
<td>Virtual Address Space Access Control</td>
<td>[142]</td>
</tr>
<tr>
<td>Virtual Address Space Structure Selection</td>
<td>[154]</td>
</tr>
</tbody>
</table>

Table 5.9 – This table presents excluded multiply described patterns which are only described once or twice after removing patterns published by Yskout et al. [540]. Multiply described patterns depicted by Yskout et al. [540] are marked with a grey background.
can always be used to enhance the description of the first published pattern. In a first step, we remove pattern descriptions by Yskout et al. [540] from the pattern list.

Furthermore, on calculating the description length of the selected patterns, it has been observed that the described patterns by Yskout et al. [540] and Blakley et al. [50] are much shorter than the others. On looking more closely, Blakley’s Authenticator pattern is not described. They remark that the pattern refers to another pattern in their catalog, but the pattern will be investigated in upcoming pattern workshops. Moreover, the multiply described patterns by Fernandez [144] have almost the same set of used sections as the ones described by Schumacher et al. [437]. The description form of the two books only differs in the Also Known As section, which is only used by Schumacher et al. [437]. Since the application of the section alignment cannot be demonstrated, due to strong similarity, the multiply-described patterns of Fernandez [144] are also excluded.

Finally, there are two remaining baseline patterns: Authenticator and File Authorization. Each baseline pattern has two comparison patterns with a comparable word count size (Table 5.11).

Table 5.10 – This table lists all remaining patterns after applying the defined preselection rules, e.g., at least three descriptions of a pattern must exist. For a better overview, multiply described patterns by Yskout et al. [540], Blakley et al. [50], and Fernandez [144] are marked with a grey background as excluded pattern descriptions.

<table>
<thead>
<tr>
<th>Security Pattern Name</th>
<th>1st</th>
<th>2nd</th>
<th>3rd</th>
<th>4th</th>
<th>5th</th>
</tr>
</thead>
<tbody>
<tr>
<td>Authenticator</td>
<td>[270]</td>
<td>[150]</td>
<td>[50]</td>
<td>[437]</td>
<td>[144]</td>
</tr>
<tr>
<td>Authorization</td>
<td>[151]</td>
<td>[437]</td>
<td>[144]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Controlled-Object Factory</td>
<td>[150]</td>
<td>[437]</td>
<td>[540]</td>
<td>[144]</td>
<td></td>
</tr>
<tr>
<td>Controlled-Object Monitor</td>
<td>[150]</td>
<td>[437]</td>
<td>[540]</td>
<td>[144]</td>
<td></td>
</tr>
<tr>
<td>Controlled-Process Creator</td>
<td>[150]</td>
<td>[437]</td>
<td>[540]</td>
<td>[144]</td>
<td></td>
</tr>
<tr>
<td>Execution Domain</td>
<td>[142]</td>
<td>[437]</td>
<td>[144]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>File Authorization</td>
<td>[151]</td>
<td>[142]</td>
<td>[437]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multilevel Security</td>
<td>[151]</td>
<td>[437]</td>
<td>[144]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Role-Based Access Control (RBAC)</td>
<td>[151]</td>
<td>[437]</td>
<td>[144]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5.11 – The pattern comparison setup. A grey background marks the usage of the section alignment for the comparison process.
5.4.4 Object Preparation

The oldest publication (baseline) of a pattern is compared to a later published one (comparison pattern). Then the multiply-described patterns are divided randomly into two sets. For one set (marked gray in Table 5.11) the developed section alignment (Table 5.3) is available for the comparison. On inspecting the other set, no additional section mapping is used.

5.4.5 Comparison Process

Later on, the prepared security patterns are compared by the author. First of all, the baseline pattern is read and then it is compared against the first unaligned comparison pattern. After that, the aligned pattern description is used for the comparison. The time is measured for each comparison.

By contrasting the patterns with different methods, the following two questions must be answered:

- Does the comparison pattern describe the same security issues as the baseline pattern? (yes, no)
- What are the differences between the described patterns?

5.4.6 Analyzing the Results

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Baseline Pattern</th>
<th>Comparison Pattern</th>
<th>Comparison Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Authenticator</td>
<td>[270]</td>
<td>[150]</td>
<td>18 min</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[437]</td>
<td>10 min</td>
</tr>
<tr>
<td>File Authorization</td>
<td>[151]</td>
<td>[142]</td>
<td>7 min</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[437]</td>
<td>10 min</td>
</tr>
</tbody>
</table>

Table 5.12 – Time measuring results. A grey background marks the usage of the section alignment for the comparison process.

The following sections depict the identified differences and similarities during the comparison process in depth.

Authenticator

The descriptions of Brown [270] and Fernandez [150] differ in the level of detail. The Authenticator by Brown is described in more detail. For example, it contains code examples and detailed implementation hints. When comparing their description and figures, we identified that they describe the same security issue. Brown’s publication describes in depth how the authentication is processed and which subjects (e.g., a user) and objects (e.g., a service) are related to this process. Fernandez’s and Schumacher’s pattern descriptions use seven and four sentences,
5.4 – Case Study

(a) Authenticator pattern described by Brown et al. [270].
(b) Authenticator pattern described by Fernandez and Sinibaldi [150].
(c) Authenticator pattern described by Schumacher et al. [437].

Figure 5.4 – UML class diagrams of the compared Authenticator patterns.

respectively, to describe the participants in the given figures and the authentication process [437]. Some participants of the authentication process differ in their naming such as “requestor”/“user” and “proof of identity”/“protected object” (Figure 5.4). The returning process of the “proof of identity”/“protected object” is described differently. Brown states that the “requester” object acquires the “protected object”. However, Fernandez and Schumacher describe that a “handler” returns the generated “proof of identity” to the “user”.

Fernandez’s description provides only additional information in the Known Uses section where more concrete software applications are mentioned. Schumacher provides additional information in the sections Problem, Consequences, Known Uses and Related Patterns.

On contrasting the three patterns, it has been observed that Fernandez’s publication and Schumacher’s share description parts and use the same figures to describe the Authenticator pattern. It seems that Schumacher’s description is an enhanced version of Fernandez’s authenticator pattern.

File Authorization

All File Authorization descriptions provide the same class diagram. The comparison publications additionally contain a sequence diagram in the Dynamics section and describe the authorization process with the given objects. Moreover, they present the consequences of applying the pattern at large. The Known Uses section mentions the same application occurrences of the pattern. Fernandez’s second publication of the File Authorization pattern [142] gives additional information, which other patterns can be relevant to applying the pattern. On contrasting the aligned Forces section of the base to Schumacher’s description, the base description requires explicitly that every subject has a home directory on the workstation for applying this pattern. Moreover, Schumacher illustrates the necessity of the pattern by a trivial example in the Example and Example Resolved sections.
On contrasting the three patterns, it has been observed that all three descriptions share textual parts and use the same class diagram to describe the File Authorization pattern. The base pattern is enhanced by Fernandez’s description published in 2002, which is also enhanced by Schumacher’s version of the File Authenticator pattern.

5.4.7 Resumé

All in all, the compared File Authorization and Authenticator descriptions specify the same security issue and same solution. The comparison descriptions provided in some cases additional information that could be used to enhance the base pattern description. The section alignment allows one to compare the content in depth and identify with regard to the content the little differences. Moreover, the comparison with aligned sections is better supported than without the mapping (see Table 5.12). With regard to the case study hypothesis, it can be retained that using the section mapping on comparing security patterns is more efficient than not using it. Finally, we have seen that the description quality differs among the multiply-described patterns and the developed section mapping is applicable to identify differences between pattern descriptions.

5.5 Summary

In this chapter, the additionally used sections of software-security pattern templates were identified by comparing them to the common POSA and GoF pattern templates. We have seen that security-pattern descriptions often have additional sections, and most of them use a POSA-like description style. Moreover, section definitions were collected and discussed whether these additional sections are security-specific or not. Regarding Question 1 it can be concluded that the added sections are not security-specific.

Approximately 53% of the software-security publications do not use any additional section to describe security patterns. Sections based on the POSA or GoF template are more common to describe software-security patterns. Based on the result mentioned above, the additionally used sections are analyzed whether they provide new characteristics of a pattern or describe common sections with a different name. Many security description sections can be aligned with common GoF/POSA sections. Moreover, the added sections that could not be mapped to common sections are used in only a few publications. Thus, the answer to Question 2 is that a security pattern does not need further significant sections to describe its security relevance or issue, but the security potential of a security pattern can be stressed by using the Forces section.

Concluding, the heterogeneity of description forms is not essential to describe software-security patterns. Based on the analyzed description-template landscape, existing sections of POSA and GoF are sufficient to present a software-security pattern, and the additionally used sections can be mapped to them. We have tested the section mapping with multiply-described security patterns and shown that it
is an efficient way to compare the heterogeneously described security patterns. Our mapping can be used to consolidate pattern information to obtain better-documented patterns or to compare patterns in depth. In case new sections are used in the future to describe security patterns, they can be added to the mapping.

Despite the developed mapping, a base set of sections for software-security patterns is eligible to avoid the increase of section heterogeneity. Moreover, the quality of descriptions can be increased by having base sections that should be discussed in a software-security pattern presentation to avoid missing information.
CHAPTER SIX

SOFTWARE-SECURITY PATTERNS: DEGREE OF MATURITY

Since Gamma et al. published their design patterns, patterns are popular in the area of software engineering [173]. They provide best practice to handle recurring problems during the software development phase. Three years later, security patterns appeared and provided solutions for security problems [538]. Besides the name analogy, design and security patterns should be very similar in their nature except for the security factor. In research and industry software engineering with design patterns is widespread. However, some researchers suspect that security-pattern engineering is made difficult by some issues such as terminology or description form, e.g., de Muijnck-Hughes and Duncan [103]. Detecting adoption problems can help to improve security patterns in the future. Thus, they can promote the awareness of security especially in the software maintenance phase where many programmers first give attention to security problems. Therefore, this chapter compares the well-explored design and security patterns to find indicators for negative impact on security pattern engineering in software development. The thereby identified maturity degree of security patterns can provide further research opportunities on security patterns and additional insights for their recognition in source code.

6.1 Introduction

Depending on one’s point of view, there may be obvious similarities between design and security patterns. This possibly refers to the name analogy or implied properties if one is thinking about patterns in software development. The personal background often forms assumptions of pattern properties. Therefore, it is possible that one associates security patterns with securing enterprise processes or network infrastructures. Possibly, software developers may link Gamma et al.’s popular design patterns to security patterns, which are close to source code and design decisions [173]. As far as we know, no work captures how close or diverse design and security patterns are.
The GoF design patterns are well-explored and research investigations are still going on. This is shown, for example, by Ampatzoglou et al. [20]. They investigate the state-of-the-art of GoF patterns by collecting case studies between the years 1995 and 2010. Then, they extract research subtopics, their activity state and the reported effect of GoF patterns on software quality. All in all, they show that all research subtopics are still active and rate pattern detection as the most active one. GoF patterns are also taught at courses to programmers as good structure and common best practice during the software development [468]. Thus, many software developers know design patterns and apply them in software systems [211].

Some researchers denote differences between design and security patterns [503]. Moreover, Muijnck-Hughes and Duncan present issues that can affect the engineering of security patterns negatively [103]. Based on their experience in working with security patterns they give the following issues in their work: pattern development, pattern templates, pattern encoding, pattern classification, pattern repositories, and pattern evaluation. However, they do not provide concrete evidence that these issues have a negative impact on pattern adoption. Detecting adoption problems can help to improve security patterns and their descriptions in future. Thus, they can promote the awareness of security, specifically in the software maintenance phase where many programmers first give attention to security problems. Therefore, the state-of-the-art in research with regard to security and GoF design patterns is captured to find gaps that may indicate problems in adopting security patterns.

Some indicators have been found stating that the state-of-the-art of both pattern types varied. For example, VanHilst and Fernandez suggest that “security patterns, . . . , are larger than GoF patterns” and “differences between GoF patterns and security patterns could play a role in pattern detection” [503]. Therefore, design and software security patterns are compared with regard to issues that can hamper their adoption along the software life-cycle.

This chapter focuses on the topics terminology, classification and description form to capture the state-of-the-art in applying patterns. Terminology and classification depict which support users have at hand by finding a suitable pattern (family) for a problem. The description form of a pattern should help a user understand the presented best practice for a problem that is necessary for later implementation. The last aspect compares the research efforts to indicate possible gaps. These aspects are assembled to capture the maturity level of software-security patterns based on a comparison to design patterns. The maturity of software-security patterns can lead to further research questions based on the detected gaps.

Our focus lies on the common design patterns published by Gamma et al. [173]. Due to their usual usage in research activities [20] and industry and open source software [19, 390, 525], they are regarded to be a good benchmark base.

However, not all security patterns deal with issues targeting the software development. Thus, the basis for this chapter are the security patterns of the Software domain, which describe mostly how to structure software parts to meet security requirements (Section 4.4). Some of the collected patterns are multiply described according to the pattern name, e.g. the Authenticator pattern, which has been found
five times [50, 144, 150, 270, 437] (Section 3.1.2). Therefore, only the first publication of patterns with the same name is used to build the base set. This chapter will use these 195 patterns as the basis to compare software-security patterns against the GoF design patterns to determine the maturity of them.

Figure 6.1 – The structure of Chapter 6.

Figure 6.1 depicts the structure of this chapter. The next section contrasts the application of design patterns to software-security patterns with regard to the software-development cycle including their considerations in the research community. This is followed by a terminology examination in Section 6.3.1 and a discussion of published classifications in Section 6.3.2. In the following section, we inspect the pattern description forms focused on used description sections, UML diagrams and code examples. Finally, Section 6.3.4 gives a brief resume of the identified issues on the pattern adoption and Section 6.4 summarizes the determined maturity degree of the software-security patterns.

6.2 Patterns and the Software Life-Cycle

This section contrasts the application of design and security patterns to determine application gaps that can be indicators for security-pattern adoption problems. The research activities on security and design patterns in forward engineering and maintenance activities are investigated. Thus, similar to the literature review presented in Chapter 3, a literature survey was conducted. We started the search with common pattern-related conferences like PLoP or EuroPLoP and inspected additionally the IEEE and the ACM Digital Library [1, 256].

The conferences and electronic publications of the years 1994 to 2013 were skimmed for several keywords, such as software engineering, reengineering, redocumentation, maintenance and refactoring, in combination with design or security patterns. A search produced much output, only the first 100 results presented at each search page were considered due to increasing number of irrelevant publications after 100 results.

At first, all publications that contain these keywords were picked out, secondly we read the abstract to find whether it described any form of research concerning the software life-cycle-like case studies. As a next step, the publications were read and divided into the fields of forward and reverse engineering. Finally, the cross-references mentioned in these documents were checked on presenting other publications of our
interest. The following sections illustrate the usage of design and security patterns in forward and reverse engineering activities.

### 6.2.1 Software Engineering—Forward Engineering

According to Chikofsky et al., forward engineering is “a traditional process of moving from high-level abstractions and logical, implementation-independent designs to the physical implementation of a system” and consists of three major phases [82]:

1. Requirement Collection—This phase collects constraints and defines the problem being solved with the software system.
2. Design—The solution is defined during the design phase.
3. Implementation—The implementation phase embraces implementing the solution, testing it and delivering the system.

The usage of **design patterns** has some advantages with regard to forward engineering. First of all, they have a positive impact on software quality [280]. The software is better structured and can be more easily modified [129]. Briefly spoken, they enhance the quality and maintainability of software [370, 371]. In addition, the defined and named paradigm allows developers to communicate these specific software interactions [228]. They isolate the variability that may exist in the system requirements, making the overall system easier to understand and maintain. Second, design patterns make communication between designers more efficient. Moreover, a software architecture can be generated by patterns as described by Beck and Johnson [38]. This gives one the opportunity to examine a software’s architecture and obtain a clear view of the design decisions the designer made.

**Security patterns** are said to be used primarily in forward engineering phases, e.g. [14, 428, 503, 538]. Many papers discussing security patterns and requirements engineering exist, e.g. [102, 230, 406, 471]. This also applies to the design [115, 357, 405] and implementation phase [50, 437, 464].

Some researchers have shown the impact of using security patterns in software systems by conducting case studies. Hafiz et al. have inspected the architecture of mail transfer agents. In particular, they compare the architecture of *qmail* in terms of an efficient security pattern usage in contrast to *sendmail* [208]. Later, they stress some evolutionary aspects in terms of *sendmail* X [202]. Karppinen et al. have conducted a case study with a software architecture visualization and evaluation tool to detect a security back door [276]. This back door originated from an inaccurately implemented security pattern. For the case study, the *Check Point* security pattern was completely removed from the software system under examination. In the end, they compared the tool-generated architectures (with and without security patterns) and showed that applying the pattern assured the security of the software system.

Halkidis et al. have shown that the use of security patterns can offer reasonable protection against most common attacks [214]. They built a small JEE e-commerce application with and without security patterns and checked common attacks against
the two implementations. These works focus on the hypothesis that security patterns improve the security of a software system, however, they do not regard points like better software quality or improved comprehension through applied patterns. These aspects should also be investigated in future.

Design and security patterns have some aspects in common with regard to forward engineering. They present a proper solution for a specific problem and support the communication among requirement engineers, designers and programmers through giving a common problem a name. As highlighted before, the use of patterns can have a positive effect on the implemented software. Design patterns can increase the maintainability and quality [280, 370, 371] and security patterns can increase the provided security level of a software system [208, 214, 276].

6.2.2 Software Maintenance—Reverse Engineering

The first way to develop a software system is to use forward engineering tools and methods, such as defining requirements, software architecture design and implement the software. This initial development is only the smallest part within a software’s life-cycle [135, 167]. Most of the work is done after a software has been released. Thus, software maintenance is an essential task in a software’s life-cycle. Often a software architecture is designed by using patterns. Hence, maintenance, in general, has two main motivations to deal with patterns:

1. Redocumentation and Design Rediscovery—Software architecture recovering to support program comprehension.

2. Restructuring—Recognition of poorly implemented code fragments to remove or optimize them with refactorings to increase the software quality (e.g., performance, security).

In most maintenance scenarios, no or inaccurate documentation exists, or developers think that in any case the best and actual documentation is the source code. Hence, the starting point on reverse engineering a software project is to redocument and recover the existing design. Afterward, the software is restructured according to the desired goal, such as better maintainability or adapting to new libraries or framework releases. In the following sections, these maintenance motivations with respect to design and security patterns are presented.

Redocumentation and Design Rediscovery

Primarily, design patterns were said to be used in a software’s design phase. Later on, some researchers, however, consider them for the topic of reverse engineering and program comprehension. A good overview of pattern detection approaches with regard to automatic and interactive software design recovery give Dong et al. [112, 113] and Rasool and Streitferdt [379].

Gravino et al. determined the effect of documented design-pattern instances on the comprehension of source code [194]. They showed with their controlled
maintenance experiment that the program comprehension of a software system is better when design pattern instances are properly documented and provided with the source code rather than non-documented instances.

In 2007, VanHilst et al. depicted that security patterns can also be interesting for reverse engineering activities, specifically for detecting them in existing software systems [503]. A first step towards improving the software comprehension by recovering security patterns in the software architecture was presented in 2011. We have shown how security patterns in code can be detected with the help of the reverse engineering tool-suite Bauhaus (see further Chapter 7). Two other approaches conduct the design recovery manually without the help of a tool that supports program comprehension by increasing the abstraction level of the software [202, 208].

Restructuring

Restructuring a software system is also called refactoring. This reengineering activity is a transformation of the software within the same level of abstraction. It should preserve the same level of functionality and semantics.

Manual and automated refactorings of the GoF design patterns are a well-explored area of research in software engineering, e.g. [86, 171, 279]. Therefore, today many integrated development environments (IDE) for programmers have plug-ins for automatic refactorings such as “rename” and “encapsulate field” [358, 481].

Using security patterns for restructuring software is not a common field of research, yet. Hafiz et al. describe an approach to use security patterns for software architecture transformations [210]. They depict how program transformations can be applied to remove security vulnerabilities in software architectures to improve the security of software systems. These “refactorings” have a slightly different goal from known design pattern refactorings. Refactorings of design patterns preserve the program behavior. In contrast to that Hafiz’ refactorings make changes to the behavior of the software system by adding security features.

6.2.3 Resumé

Researchers cover almost all directions in the software’s life-cycle (forward/reverse engineering) for design and software security patterns. The exploration of the positive impact of using software-security patterns to mitigate common security threats is comparable to the quality and maintenance enhancement of using design patterns for designing and implementing a software system. However, more investigations on software-security patterns are desirable.

In the area of redocumentation, the design recovery of design patterns is well-explored. Many tools can be found that support a maintenance programmer in finding design patterns in code. In contrast to that, no semi-automatic or interactive support for software-security patterns has been found.

Manual and automated refactorings for design patterns are well proven and explored. In contrast, automated refactoring for security patterns is a new direction in security research. Moreover, to the best of our knowledge, no studies about
applying security refactorings manually exist. Also, studies on how the usage of security patterns can support the software comprehension are eligible.

6.3 Possible Issues Affecting the Adoption of Patterns

Possibly, the specific security target of security patterns makes them less popular in some research areas. According to Mujinck-Hughes and Duncan other aspects can also have a negative impact on their adoption [103]. They suggest that the aspects development, encoding, templates, classification, repositories, and evaluation can hamper the security-pattern engineering. This section focuses on issues that can have negative influences on the adoption of security patterns from a developer’s point of view. Thus, the issues pattern development and encoding will be left out. Due to the well-discussed repository aspect for design and security patterns [103, 236, 320] and the only repository of the GoF patterns being available for comparison, this issue will also not be discussed here. Investigated are the issues terminology, classification and their presentation for design and security patterns to determine possible divergences. These issues can support a developer by selecting and applying the appropriate pattern for a problem at hand. Therefore, they can be regarded as main topics on pattern usage in the software life-cycle.

6.3.1 Terminology

First of all, on working with patterns one must know the terms in a field or domain to be able to search and select suitable concepts for a given problem. This procedure can be supported by clearly defined terms and their reuse across different venues. We have seen in Chapter 4 that security patterns can be assigned to different domains and thereby they can be possibly assigned to different terms. Thus, the difference of the terms “design patterns” and “security patterns” are inspected.

Design patterns have been proposed as specific solutions for recurring problems in software design [173]. Due to their focus on software development, these patterns are also often called software design patterns.

Yoder and Barcalow introduced the term security patterns for patterns targeting security [538]. However, a problem of the “security pattern” term is, “there exists no clear definition of a security pattern, since different authors refer to security patterns in different contexts.” [213].

Sometimes, security patterns may be reflected directly in software. In this case, security patterns typically have an impact on design and architecture of software systems. These patterns will also be called software security patterns or security design patterns, which implies an alikeness to the design patterns [300, 526]. For example, Ramachandran refers to security patterns as basic elements of system security architecture [378]. Some authors also reduce the term security pattern
to only software-related problems, but they still call them security patterns, e.g., Yoshioka et al. [539].

Other authors use the term “security pattern” for their special-purpose patterns [101, 136, 404]. They can address administrative issues for securing software, e.g., XML Firewalls or with a simultaneously running Hot Standby backup system mirroring important data and services [107, 208]. Moreover, the term can be used for patterns that affect activities of users, enterprises or other software life-cycle phases different from the design, e.g., Keep It Secret [391], Increased Monitoring For Intellectual Property Theft By Departing Insiders [339], or Access Control To Physical Structures [155].

On contrasting software design and security patterns, there can be a problem of terminology. This can be a result of the aforementioned diverse application domains that security patterns address.

### 6.3.2 Classification

A classification organizes objects, e.g., patterns, into groups that share one or many properties, such as a physical nature or a particular purpose. Since it is impractical to look at all details of patterns during pattern selection, such an arrangement provides an overview of a large number of patterns. Moreover, classification criteria can help to understand the nature and value of a pattern. All in all, a classification can support the user in selecting patterns [541]. Thus, the state-of-the-art on classifications is briefly depicted.

Section 4.2 collects published design and security pattern classifications and gives an extensive description and overview of them. These classifications share similarities, but also have differences, which were discussed extensively. The latest classification was published in 2011 by [14].

We have seen in Section 4.2.3 that an evolution of design and security-pattern classifications exists with respect to the reused classification concepts in general. The influence among security and design-pattern classifications is shown in Figure 4.7. This influence is often expressed by reusing classification aspects, e.g., the structural aspect of Gamma et al.’s classification [173] are reused by security classifications, such as Kienzle et al. ’s and Konrad et al. ’s classifications [282, 291].

Finally, the research activities with regard to the classifications on security patterns are still ongoing. The latest activities are done by Alvi and Zulkernine [15]. They compared the existing security pattern classifications with regard to a desirable classification quality metrics [220] and provide a bird’s-eye view on them. As far as we know, such studies have not been conducted in the last years for design pattern classifications neither new classifications have been published since 2006. Moreover, almost every year a new classification approach appears for security patterns (Section 4.2.2 and Alvi and Zulkernine [15]). According to the evolution and assessment the maturity degree of security-pattern classifications is better than the design patterns.
6.3.3 Pattern Presentation

“A description model has to provide enough detail and guidance for developers to incorporate them in their applications.” Fernandez et al. [158]

The presentation of a pattern is an important attribute. It provides the possibility to determine whether a pattern is applicable to a particular situation and how the actual instantiation of a pattern should be done. Some works have already covered the description quality [238, 300, 539]. This point is discussed in the following section. Moreover, further aspects that can influence the description quality and thereby understanding and adoption of patterns are additionally worked out.

Description Form

The authors of the GoF patterns provide an extensive description for every design pattern [173]. They also use code examples and UML diagrams to highlight pattern characteristics and basics. Based on this book many other descriptions (books, websites, blogs) can be found for the GoF patterns, which sometimes provide additional information on a specific pattern. For example, on the Wikipedia website the Singleton pattern description provides five different Java implementations and the Composite pattern code examples in Java and C# to illustrate the pattern [527]. Therefore, it can be assumed that the documentation state for GoF patterns is quite well.

Security patterns do not provide such a good documentation state. Some works have depicted that the quality of security pattern descriptions differs and is sometimes not sufficient. For example, Laverdiere et al. utilizes the House of Quality evaluation framework to evaluate patterns [300] and show that the twelve inspected patterns have many undesirable properties such as lacking generality, or under- or over-specification. The appropriateness and quality of documentation has been examined by Heyman et al. [238]. They use a scoring system to measure the quality of a description element and show that core patterns have medium to high quality. In addition, the guidelines, best practices, and process activity patterns have a documentation quality between low and medium. However, no detailed information is given. Yoshioka et al. inspected patterns according to a pattern’s ease of use, effectiveness, and sufficiency [539]. Unfortunately, they only examine a few patterns according to ease of use with regard to their description form.

All mentioned evaluations take only a small subset of security patterns into account and do not evaluate the usage of UML diagrams or code examples. However, the comprehension and adoption of concepts can be increased by using figures and code examples. Therefore, sections used for pattern description, UML diagrams, and code samples are considered as further aspects to determine the software-security pattern maturity level.
Heterogeneity of Description Aspects

The GoF book provides a unique description form for the presented patterns. This enables the user to compare the patterns and choose the right pattern for a problem easily.

In contrast, software-security patterns are also described uniquely within a single publication, but across the different publications the used description aspects vary (Section 4.3.1). Different publications use different section names to describe the same pattern characteristic. One example is the Consequences section. Some publications use this section as it is and others use Trade-Off, Liabilities, or Security Factors & Risks instead (Chapter 5). For this reason, the reader must know which sections can be mapped to contrast security patterns that are not depicted in a single publication. Otherwise, this heterogeneity may hamper the adoption of the right pattern for a security problem to be solved.

Security patterns should be presented in a more unique form to ensure a high quality of pattern description to avoid under- and over-specified patterns [300]. An early approach to a unique description in combination with a pattern repository depict Maña et al. [320]. Actually, published security patterns have no unique description, which makes the selection and comparison difficult.

UML Diagrams

The usage of additional items such as diagrams in software-development related documents supports the understanding of software and the security pattern adoption [25, 238]. Thus, the use of graphical representation, such as UML diagrams, is one qualitative criterion for pattern descriptions [222, 238]. The clear shape of a pattern is often expressed by the usage of UML diagrams and code samples. They indicate which classes and their relationships are indispensable for the pattern’s functionality. This essential structure is also called a micro-architecture that describes the structure with a subset of classes, methods, and fields [196, 199].

MySingleton
- instance: MySingleton
- MySingleton(): MySingleton
+ get Instance: MySingleton

Figure 6.2 – The Singleton micro-architecture.

An example is the Singleton pattern’s micro-architecture [173]. This pattern consists of a single class that has a private constructor and a getInstance method. This method controls the class instantiation by returning the former created class instance. Moreover, the class contains a variable of the class type which holds the once created class instance (Figure 6.2).

In contrast to the documentation mentioned above, security patterns are mostly described abstractly [238]. Only some of them contain figures and source code
snippets to clarify the software-security pattern modeling during the software design phase. Possibly, these circumstances have an impact on the software design when one must ensure security objectives and choose the best security pattern for the software needs.

The set of software-security patterns and the aforementioned design patterns of Gamma et al. [173] were inspected to measure the quantity of UML diagram usage in pattern descriptions. The result is shown in Table 6.1.

Almost all design patterns are described with UML class diagrams and one fourth additionally with UML sequence diagrams. In addition, all of them are sketched with code examples, but no activity or use case diagrams are used. On the security pattern side, 59 percent of the patterns have class diagrams and 42 percent sequence diagrams. Rarely, security pattern descriptions are enhanced by activity (2 %) or use case diagrams (1 %).

<table>
<thead>
<tr>
<th>Documentation Type</th>
<th>Used by Design Patterns</th>
<th>Used by Security Patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td>UML Class Diagram</td>
<td>100 %</td>
<td>59 %</td>
</tr>
<tr>
<td>UML Sequence Diagram</td>
<td>27 %</td>
<td>42 %</td>
</tr>
<tr>
<td>UML Activity Diagram</td>
<td>0%</td>
<td>2 %</td>
</tr>
<tr>
<td>UML Use Case Diagram</td>
<td>0%</td>
<td>1 %</td>
</tr>
<tr>
<td># of Analyzed Patterns</td>
<td>23</td>
<td>195</td>
</tr>
</tbody>
</table>

Table 6.1 – Additional documentation type usage.

**Code Examples**

“The best way to understand design patterns is by seeing them in action.” Gamma [172]

To apply patterns during design, implementation or maintenance phases, they should contain precise and comprehensible examples to be applied all over the software life-cycle. Having a look at design patterns, they can be found in several types of software and are a base of a well-defined code example set in software development. For example, if one searches with the help of Google for the Singleton pattern, right on the first result site one get several pages with UML diagrams and many code examples as that one from Wikipedia [527].

Newer design patterns do not only provide an abstract text describing the patterns, but also UML diagrams and comprehensive code examples to stress the implementation of the pattern. This additional information makes them more understandable for programmers and software designers, e.g., Austrem’s Mix’n’Match pattern [29]. Common design pattern books do so as well [13, 72, 173].

Therefore, this section focuses on the provided code examples of software-security patterns. 41 code examples are depicted in the aforementioned software-security pattern set. This indicated that only 21 percent of the patterns provide code
examples. The majority of these code examples—20 code examples—are from the JEE security pattern book by Steel et al. [464].

<table>
<thead>
<tr>
<th>Security Pattern</th>
<th>Programming Language</th>
<th>Compiles</th>
<th>Code Provided by Description</th>
<th># SLOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Authenticator [270]</td>
<td>Java</td>
<td>✓</td>
<td>✓</td>
<td>51</td>
</tr>
<tr>
<td>Assertion Builder [464]</td>
<td>Java</td>
<td>✓</td>
<td>✗</td>
<td>200</td>
</tr>
<tr>
<td>Audit Interceptor [464]</td>
<td>Java</td>
<td>✓</td>
<td>✓</td>
<td>72</td>
</tr>
<tr>
<td>Authentication Enforcer [464]</td>
<td>Java, XML</td>
<td>✓</td>
<td>✗</td>
<td>25+17</td>
</tr>
<tr>
<td>Authorization Enforcer [464]</td>
<td>Java, JSP</td>
<td>✓</td>
<td>✗</td>
<td>35+75</td>
</tr>
<tr>
<td>Container Managed Security [464]</td>
<td>Java, XML</td>
<td>✓</td>
<td>✓</td>
<td>12+71</td>
</tr>
<tr>
<td>Credential Tokenizer [464]</td>
<td>Java</td>
<td>✓</td>
<td>✗</td>
<td>84</td>
</tr>
<tr>
<td>Dynamic Service Management [464]</td>
<td>Java</td>
<td>✓</td>
<td>✓</td>
<td>155</td>
</tr>
<tr>
<td>Interception Validator [464]</td>
<td>Java</td>
<td>✓</td>
<td>✗</td>
<td>24</td>
</tr>
<tr>
<td>Interception Web Agent [464]</td>
<td>XML</td>
<td>✗</td>
<td>✓</td>
<td>58</td>
</tr>
<tr>
<td>Keep Session Data in Client [475]</td>
<td>HTML</td>
<td>✗</td>
<td>✓</td>
<td>5</td>
</tr>
<tr>
<td>Keep Session Data in Server [475]</td>
<td>HTML</td>
<td>✗</td>
<td>✓</td>
<td>4</td>
</tr>
<tr>
<td>Message Inspector [464]</td>
<td>Java</td>
<td>✗</td>
<td>✓</td>
<td>61</td>
</tr>
<tr>
<td>Multiple Secure Observers using J2EE [180]</td>
<td>Java</td>
<td>✗</td>
<td>✓</td>
<td>159</td>
</tr>
<tr>
<td>Obfuscated Transfer Object [464]</td>
<td>Java</td>
<td>✗</td>
<td>✓</td>
<td>25</td>
</tr>
<tr>
<td>Password Synchronizer Pattern [464]</td>
<td>Java</td>
<td>✓</td>
<td>✗</td>
<td>901</td>
</tr>
<tr>
<td>Role Based Fine Grain Access Control [409]</td>
<td>Java</td>
<td>✗</td>
<td>✓</td>
<td>11</td>
</tr>
<tr>
<td>Policy Delegate [464]</td>
<td>Java</td>
<td>✓</td>
<td>✗</td>
<td>94</td>
</tr>
<tr>
<td>Secure Based Action [464]</td>
<td>Java</td>
<td>✓</td>
<td>✗</td>
<td>24</td>
</tr>
<tr>
<td>Secure Logger [464]</td>
<td>Java</td>
<td>✓</td>
<td>✗</td>
<td>37</td>
</tr>
<tr>
<td>Secure Pipe [464]</td>
<td>Java</td>
<td>✓</td>
<td>✗</td>
<td>49</td>
</tr>
<tr>
<td>Security Policy [318]</td>
<td>Java</td>
<td>✓</td>
<td>✓</td>
<td>47</td>
</tr>
<tr>
<td>Secure Service Facade [464]</td>
<td>Java</td>
<td>✗</td>
<td>✓</td>
<td>40</td>
</tr>
<tr>
<td>Secure Service Proxy [464]</td>
<td>Java</td>
<td>✗</td>
<td>✓</td>
<td>69</td>
</tr>
<tr>
<td>Secure Session Object [464]</td>
<td>Java</td>
<td>✗</td>
<td>✓</td>
<td>10</td>
</tr>
<tr>
<td>Session Scope [475]</td>
<td>Java</td>
<td>✓</td>
<td>✓</td>
<td>25</td>
</tr>
<tr>
<td>Session Timeout [475]</td>
<td>Java</td>
<td>✓</td>
<td>✓</td>
<td>21</td>
</tr>
<tr>
<td>Single Sign-on Delegator [464]</td>
<td>Java</td>
<td>✓</td>
<td>✗</td>
<td>129</td>
</tr>
</tbody>
</table>

Table 6.2 – Results of code example inspection for security patterns.

Collecting the Examples In the majority of cases, the pattern examples were reconstructed by copying the code snippets within a pattern description to separate files. This technique was applied to the code examples of several works [115, 180, 270, 318, 475]. Fortunately, Steel et al. provide the most of their code examples in a downloadable ZIP file [463]. Some examples, however, were also copied from the book because they are missing in the ZIP file.

The GoF code examples are also available for download [268]. The provided examples are slightly different from the book examples. They also contain the
code snippets of the implementation and known use aspects, which were additional explanations of the pattern or a specific variant. Moreover, missing classes from the description were added to the implementations.

<table>
<thead>
<tr>
<th>Security Pattern</th>
<th>Programming Language</th>
<th>Compiles</th>
<th>Code Provided by Description</th>
<th># SLOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clear Sensitive Information [115]</td>
<td>C++</td>
<td>✓</td>
<td>✓</td>
<td>113</td>
</tr>
<tr>
<td>Defer To Kernel [115]</td>
<td>C++</td>
<td>✓</td>
<td>✓</td>
<td>147</td>
</tr>
<tr>
<td>Input Validation [115]</td>
<td>C++</td>
<td>✓</td>
<td>✓</td>
<td>59</td>
</tr>
<tr>
<td>Pathname Canonicalization [115]</td>
<td>C++</td>
<td>✓</td>
<td>✓</td>
<td>7</td>
</tr>
<tr>
<td>Privilege Separation [115]</td>
<td>C++</td>
<td>✓</td>
<td>✓</td>
<td>100</td>
</tr>
<tr>
<td>RAII (Resource Acquisition Is Initialization) [115]</td>
<td>C++</td>
<td>✓</td>
<td>✓</td>
<td>25</td>
</tr>
<tr>
<td>Secure Builder Factory [115]</td>
<td>C++</td>
<td>✓</td>
<td>✓</td>
<td>159</td>
</tr>
<tr>
<td>Secure Chain of Responsibility [115]</td>
<td>C++</td>
<td>✓</td>
<td>✓</td>
<td>101</td>
</tr>
<tr>
<td>Secure Directory [115]</td>
<td>C++</td>
<td>✓</td>
<td>✓</td>
<td>81</td>
</tr>
<tr>
<td>Secure State Machine [115]</td>
<td>C++</td>
<td>✓</td>
<td>✓</td>
<td>180</td>
</tr>
<tr>
<td>Secure Strategy Pattern [115]</td>
<td>C++</td>
<td>✓</td>
<td>✓</td>
<td>59</td>
</tr>
<tr>
<td>Secure Visitor [115]</td>
<td>C++</td>
<td>✓</td>
<td>✓</td>
<td>37</td>
</tr>
<tr>
<td>Unique Atomic Chunks [201]</td>
<td>C++</td>
<td>✓</td>
<td>✓</td>
<td>33</td>
</tr>
</tbody>
</table>

Table 6.3 – Results of the C++ code example inspection for security patterns.

Inspection of Examples From a developer’s point of view, patterns should provide implementation hints encoded as source code examples. Therefore, 23 design and 41 security pattern examples were inspected. All design pattern and 13 security pattern examples are written in C++. 25 security patterns provide examples in Java code and a very few are presented in HTML, XML or JSP (Table 6.2 and 6.3).

A problem for the comparison is the fact that some examples are written in Java, HTML, Python, and some in the C++ programming language. Comparing examples written in different programming and markup languages is not meaningful. Hence, some basic data of the examples provided in the pattern descriptions are collected. We refrain from translating the code examples into one program language, e.g., C++ to ADA, such action may falsify the result through the different language implementation possibilities during the transcription.

The code examples were inspected according to the following points:

- Determine whether the provided code example is compilable, thus no relevant part is missing.
- Provide code comments.
- Provide source lines of code (SLOC) (measured with the cloc tool [353]).
Compilability Only 14 of the 41 software-security pattern examples could be compiled (Table 6.2 and 6.3). The following compiling problems were observed:

- Incomplete examples (often)
- Syntactic or typing failures (seldom)

The reader can often solve syntactic problems if she is familiar with the programming language used in the example. In contrast to the completely implemented design patterns, the security-pattern code examples are often incomplete. Due to missing implementations, it is hard to use an example and understand the pattern (and its constituting parts) by running and modify it.

Comments Comments support the understanding of source code [476, 533]. Four software-security examples do not have any comments. The software-security examples have a median of ten comment lines which is lower than the median of 40 of commented lines on the GoF examples (Figure 6.3b). The maximal outlier for both sets correlates with the largest pattern example; 414 commented lines are counted in the security-pattern example with 901 SLOC and 111 commented lines in the design pattern example with 216 SLOC, respectively. This indicates, based on the median, that the quantitative annotation of source code is lower for software-security patterns compared to design patterns.

Source Lines of Code (SLOC) The tables 6.2 and 6.3 present the result for the number of code source lines for security patterns with code examples. The design...
pattern code samples consist of 55 SLOC for the Template Method pattern up to 221 SLOC for the Abstract Factory pattern. The security-pattern code samples are larger. The smallest example—the Pathname Canonicalization pattern [115]—has seven SLOC and the Password Synchronizer pattern [464] as the largest 901 SLOC. The differences may stem from the different programming languages. For a more suitable comparison, only patterns written in C++ are compared additionally. The result is shown in Figure 6.3a. Comparing 14 security pattern examples with the 23 design pattern examples, the result is that the median of both is close together 70 (security patterns) to 79 (design patterns) SLOC. In contrast to that, the scales differ. The security pattern examples vary from seven to 180 SLOC and the design pattern examples have a range of 19 to 110 SLOC with an additional outlier with 216 SLOC. Finally, the assumption of VanHilst and Fernandez that security patterns are larger than design patterns [503] cannot be substantiated by the inspected examples. More code examples for such a comparison are needed to obtain meaningful results.

6.3.4 Resumé

The description base of security patterns evolved from the pattern description in Gamma et al.’s and Buschmann et al.’s books (see also Chapter 5). Nevertheless, the high variety on description aspects is a disadvantage on comparing and selecting an appropriate pattern.

UML diagrams and code examples are used by design patterns rather than by software-security patterns to support the reader’s understanding. The understanding and realization of patterns are easier when their documentation provides assistance for the programmer or software architect. This can be positively increased by using UML documentation to depict how a pattern is shaped in code [25, 486]. At this point, the software-security patterns are not well-documented through missing UML diagrams and only a few (running) code examples can be found to implement and learn how these patterns work, but such models are necessary for an easy understanding [323]. Concluding, software-security patterns can be improved to be better understandable and provide easily adaptable solutions for programmers and software architects.

Many software-security patterns do not provide code examples and when given, many examples do not compile. This decreases the ability to testify the solution of a pattern and explore it. This deficiency can be possibly counterbalanced by the many comments most examples have. In contrast to GoF patterns, only a few software-security patterns provide code examples.

6.4 Summary

In this chapter we have presented the state-of-the-art of software-security patterns and discussed their maturity degree with regard to their classifications, description forms, code examples, and usage in the software life-cycle compared to the common design patterns. In brevity, our findings are:
The area of pattern classification is well explored for security patterns. There is ongoing research and design pattern concepts are adopted.

The area of forward engineering seems to be well-investigated for security patterns.

The documentation portion of UML diagrams in software-security patterns is low.

Only one-fifth of the software-security patterns provide code examples. The quality of examples is not as good as the design pattern examples.

In contrast to design patterns, security patterns are lesser investigated in the area of software maintenance (design recovery, redocumentation, and software restructuring).

The research on security patterns is not so widespread in the area of software maintenance. As the main reason for this observation, their heterogeneous description nature and lack of proper code examples can be seen. These aspects can affect the usability of security patterns compared to the GoF design patterns in practice and research. Possibly, further investigations on the presentation quality of security pattern and a unique description form may improve the reputation. We encourage researchers to adapt and evaluate investigated design pattern issues to software-security patterns, e.g., impact on software quality or automated tools for software redocumentation. This can promote the awareness of security especially in the software maintenance phase where many programmers give attention to security problems the first time.

The documentation quality of security patterns is lower than that of design patterns. Thus, in the next chapter, micro-architectures (UML diagrams and code examples) are deduced from existing pattern descriptions to make their architecture more concrete for further investigations.
Part III

Recognition
MANUAL SECURITY-PATTERN DETECTION

The approach introduced in this chapter was the starting point for all approaches in the area of security patterns and program comprehension presented in this thesis. As we have seen in the previous chapters, the security pattern quality varies, and its description level is quite abstract. Thus, this chapter is the initial attempt of detecting security patterns in code with the help of the reverse engineering tool-suite Bauhaus [386], based on the usage of the Resource Flow Graph (RFG) representation for software applications. This program representation enables the usage of the integrated program comprehension method called hierarchical reflexion method [292]. This method aims to reconstruct a software architecture by mapping a hypothetical architecture to the actual software architecture extracted from the source code. The objective is to depict the existence of security patterns at an abstract architectural level. Specifically, the presented approach aims to detect the Single Access Point security pattern in two case studies. Here, the security pattern is identified in a software architecture by using a program comprehension technique.

This chapter is structured as follows. In Section 7.1, the software analysis tool Bauhaus is described briefly. This section focuses on the used hierarchical reflexion method which is based on the abstract source code representation RFG. This is followed by the description of the early case studies in Section 7.2. Further steps in combining an architecture-based methodology with security pattern detection which can help programmers during the software maintenance process are discussed in Section 7.3. Finally, a summary is given in Section 7.4.

7.1 Bauhaus Tool and Hierarchical Reflexion Analysis

The Bauhaus tool-suite is a reverse engineering tool-suite that has been employed in several industry projects [386]. Bauhaus allows one to retain two abstractions from the source code. The low-level representation called Intermediate Language (IML) is an attributed syntax tree (an enhanced AST) that contains the detailed
program structure information, such as loop statements, variable definitions, and name bindings. The RFG, a more abstract representation, works at a higher abstraction level and represents architecturally relevant information of the software. At present, such a graph can be created for programs written in C, C++, Java, ADA, and C#. The RFG is a hierarchical graph that consists of typed nodes and edges representing elements like types, components, and routines and their relations. The RFG’s information is stored in structured views, where each view represents a different aspect of the architecture, e.g., a call graph.

Several analyses are built upon this infrastructure to derive design and architectural information like the so-called hierarchical reflexion analysis [292]. This analysis extends the original analysis developed by Murphy et al. [343] to hierarchical systems. It starts with a hypothesis of the architecture and a mapping of existing implementation components onto architectural components provided by a human analyst. An automated analysis then determines convergences and differences between the architecture and the implementation model, the so-called reflexion model. Based on these findings, the architecture and mapping may be refined, and the process will be repeated until the architecture model sufficiently describes the implementation.

Usually, this procedure is used when a software system has to be modified, but the documentation or the knowledge of it got lost. Besides this reconstruction, the reflexion analysis can be used to check the present implementation against its architectural specification.

The hierarchical reflexion analysis is a well-known method to reconstruct software architectures [62, 400]. We use this method to identify security patterns at an architectural level by marking potential pattern structures based on their description. Then, with an automated check against the real source code, architecture violations can be detected, such as, calling a component not through a checkpoint, which can induce security concept violations. This reflexion method has been used in the case studies presented in Section 7.2 to detect a selected security pattern in a software’s architecture.

### 7.2 Early Case Studies

This section discusses our architecture-centric security pattern analysis in the context of two case studies. The security pattern *Single Access Point* (see below) is selected to demonstrate the identification of a security pattern within an abstract software representation.

Primarily the open source instant messenger client Spark [267] and an open source Android application named Simple Android Instant Messaging Application [328] are selected. Both are Java-based programs, so the Java bytecode was used to generate the software architecture in the RFG format. This is the starting point for using the hierarchical reflexion method to detect the *Single Access Point* pattern. We present the selected pattern in Section 7.2.1. This is followed by a closer look on the case studies in Section 7.2.2 and 7.2.3.
7.2 – Early Case Studies

7.2.1 Single Access Point Pattern

A Single Access Point pattern [437] provides access to a system for external clients. Moreover, it ensures that the system cannot be damaged or misused by such clients. The idea behind this pattern is that an exclusive door to the system can be better protected and controlled than many. Figure 7.1 depicts the UML diagrams for the Single Access Point security pattern. For this reason, many application clients, such as Twitter or instant messenger clients that provide any access to other systems, use a derivative of this pattern to provide clients (mostly, users) with access to underlying services.

According to the class diagram in Figure 7.1a, a first hypothetical architecture (Figure 7.2a) is modeled. It contains the components Client, Single Access Point, and Protected System. The client that uses this software is represented by the Client component in the architecture. Given that the reflexion method is based on static code dependencies and our client in the case studies is a human being, we apparently will never see any match in the dependencies with the system. Schumacher et al. mentioned that the Boundary Protection component of a protected system is often hard to show. This also applies to our case studies where the client is a human user that needs to know a username and password to gain access to the protected system. In this case, the user’s knowledge models the Boundary Protection component. Hence, this component—as it cannot be modeled according to the usage of static analysis—it is skipped. The dependencies between components such as “calls”, “references”, and “inherits” are not specified in depth to simplify the dependencies. They are modeled as undirected dependencies according to the undirected edges in Figure 7.1a.

After the first attempt to model the architecture, we realized that the UML sequence diagram is not adequate enough to represent the idea of the Single Access Point pattern. Thus, not every direction of the dependencies between the components may be allowed to ensure a secure behavior. For this reason, a more
specific access point model according to the information given in Figure 7.1b has been created. First, the Client interacts with the Single Access Point component, and after that the client can interact with the Protected System. Based upon this, it is assumed that the Single Access Point component allows or denies the user’s request and informs the Protected System about the response. Possibly, the Single Access Point component shows a further window to allow the user to interact with the Protected System after the login. Hence, a proper behavior of the system is that the Single Access Point has dependencies to the Protected System to call, communicate or instantiate an object after passing the Single Access Point. The corresponding hypothetical architecture is shown in Figure 7.2b.

Both hypothetical architectures will be used with the hierarchical reflexion method on the chosen applications to show and discuss distinctive features.

### 7.2.2 Case Study: Spark

Spark is an open source instant messenger client that provides a login screen and is expected to use this pattern [267]. It is a client that allows users to log in to an instant messenger network and then receive and write instant messages to other users.

Intuitively, the software components are mapped according to their names such as “LoginDialog”, “LoginSettingsDialog” to the component Single Access Point. Then, the assumption is made that the rest of the code is in package “org.jivesoftware” is the Protected System.

Figure 7.3a shows the match between the hypothetical architecture and the real code architecture. As expected, outgoing and incoming edges of the Client component are marked as absence (dotted edges). The edges between Single Access Point and Protected System are marked as convergences. This shows that this pattern can be found in the software’s architecture by using the reflexion method.
7.2 – Early Case Studies

Figure 7.3 – Detection results for Spark application.

Figure 7.3b depicts the detection result of a login behavior in Spark. The architecture-match result is depicted in Figure 7.3b. The expected dependency between Single Access Point and Protected System is marked as convergence (solid edge). However, more dependencies exist than have been modeled, represented by the dashed edges. They arise from static field usages and class instantiations. This shows that the two identified components are bundled together and are not strictly separated in Spark as one might expect according to their task.

7.2.3 Case Study: Simple Android Instant Messaging Application

Based on our experiences with Spark, another open source application is examined. The Simple Android Instant Messaging Application [328] is an example application for the mobile phone platform Android [183]. The author’s intention to make this application freely available was to provide interested people with an example of an Android application and show how instant messaging can be provided easily. It communicates via the HTTP protocol with a web server. This server is also used to authenticate users for the service usage. We assume that every component starting or ending with “Login” indicates the Single Access Point pattern in the source code. The rest belongs to the Protected System because it provides the instant messaging communication with the server. For the reflexion method, the same hypothetical architectures as depicted in Figure 7.2a and 7.2b are used.

Figure 7.4b depicts the architecture-match result for the expected behavior. Here, the dependency between the components Single Access Point and Protected System is marked as convergence (solid edge) and the dependencies to the client are marked as absences (dotted edges). This indicates that in the Simple Android Instant Messaging Application the components for Single Access Point and Protected System are separated in the code. This hypothesis is confirmed by Figure 7.4a that shows an absence between Protected System and Single Access Point.
7.2.4 Resumé

These two case studies demonstrate that it is possible to detect a security pattern within a software system using the components described in the pattern description. Besides the two case studies discussed, our method is used to detect the *Single Access Point* pattern in two other software systems and to detect the *Runtime mix’n’match* design pattern [29] that is coupled with a *Check Point* security pattern. In particular, it was detected in the middleware of the open source platform Android [183].

Towards this case study, we expect to be able to analyze more security patterns. Their description should also contain UML diagrams that clarify their structure and behavior. However, shown in this study even with the help of such diagrams it cannot be clearly decided whether a security pattern is modeled accurately or inaccurately. In our case, Spark models the *Single Access Point* pattern according to the UML class diagram and the Simple Android Instant Messaging Application in compliance with both UML diagrams.

With the introduced static examination, however, it was not possible to check whether the system behaves in the expected way. Therefore, more source code information is needed. Such information is provided in the IML representation, a more specific static program description of this pattern or even dynamic analysis information. Moreover, on detecting such patterns automatically or semi-automatically, one has to deal with abstract descriptions that must be modeled differently for several application contexts. For example, in the shown case the client was a user that must enter his credentials. In another case, the client is possibly a web service that tries to use another web service.

7.3 Security Aspects and the RFG

Sohr und Berger depict some possibilities to accomplish a security analysis with the RFG [456]. This section resumes on their point and discusses other security
aspects that can be based upon the RFG. In contrast to their ideas, the presented approach in this chapter does not focus on policies and Role-based Access Control (RBAC) extensions [419]. It focuses on software quality assurance and program comprehension in conjunction with security patterns.

### 7.3.1 Security Patterns at the Architecture Level

As shown above, the RFG provides the ability to create new views. These views can be created containing only elements focusing on particular purposes. Conceivable is a view containing elements that are supposed to belong to a single security pattern, possibly identified by the method described above. If one has identified more than one security pattern and created them on different views, one can create a new view by intersecting or uniting the view to visualize composed or merged security patterns as described by Schumacher [432]. Possibly this process can be automated when the system knows several available or often occurring compositions of security patterns. Presenting such combinations to maintenance programmers may facilitate the realization of adequately and inadequately programmed pattern collaborations. For instance, consider the combination of Single Access Point and Check Point pattern [437], where a badly implemented cooperation may raise security leaks.

### 7.3.2 Automatic Detection, Suggestions, and Learning

Semi-automatic detection of security patterns is good, but it is time-consuming and requires in-depth knowledge of the system. A better approach would be the automatic detection. However, this is not easy to realize as there are many challenges as described in the previous chapters. Maybe, a pattern language exists that fulfills our needs for the description of security patterns at the architecture level. Then such a description could be transformed into the RFG model for an automatic pattern matching and can present the maintenance programmer pattern suggestions. These suggestions can be assessed or modified by the programmer to improve the security pattern model. Thereby, pattern derivatives can be collected for improving the automatic or even the semi-automatic detection. Moreover, this collection gives the abstract appearance of security patterns a more precise shape that can be reused. This technique can be refined by using security anti- or misuse patterns [61, 160] to model an architecture’s irregularities to be able to detect the incorrect usage of security patterns. The benefit of the sketched technique is that software systems can be post-checked and hardened before they will be released.

### 7.3.3 Source and Sink Markers for Pattern Endings

Information flow is an important issue concerning software security as indicated the many approaches collected by Sabelfeld and Myers [412]. A security view of the RFG can also model fractals describing the information flow. If the RFG is combined with a more-detailed code representations, such as the IML, the information flow between components can be modeled. If one has detected a security pattern or compound,
the pattern could be extracted in a new view and highlight sources and sinks of the patterns. This would enhance the role-based view described by Sohr and Berger [456], and give the opportunity to plug in a further information flow analysis to validate the pattern’s behavior. To give an example, consider the communication with a database or a password manager. In these cases, the visualization of security patterns’ sinks and sources, like architectural glue dots, highlight critical points in the source code which have to be checked carefully during reviews or maintenance activities. Such a visualization probably supports the information flow comprehension while reconstructing the software system.

7.4 Summary

This chapter has demonstrated that it is possible to detect security patterns using the reflexion method. The conducted case studies also discover some problems with the abstract description level of the security patterns. This problem has already been discussed in previous chapters within this thesis (e.g., Chapter 6). Moreover, methods that can help programmers to preserve security patterns during the software maintenance process are discussed.

The presented RFG can be possibly used in many different cases for displaying pattern information at a software’s architectural level. This information is on an abstract level for software maintainers and analysts and is of interest when the existence of security features has to be ensured for audits or documentation. This high-level view on the patterns cannot support software maintainers and analysts on checking the object configurations that are implemented to model security features in software systems. However, some works found out that the configuration of objects that form security features is the most critical issue [32, 139]. Thus, besides the fact that a software system contains security patterns, a lower level presentation of the objects that shape such a pattern is needed. Based on the discussed architectural detection and (low-level) representation of security patterns, a holistic approach is conceivable that supports analysts and software maintainers in assessing security features in software with regard to their existence and correct implementation. Therefore, the next chapter focuses on this difficulty by detecting and presenting security patterns at a level closer to source-code.
SECURITY PATTERNS IN SOURCE CODE

Detecting GoF patterns to support program comprehension is a well-explored task [20]. In contrast to security patterns, the GoF patterns are well-documented using UML diagrams and commented code examples to increase the comprehensibility of the pattern. Unfortunately, only a few security-pattern descriptions provide information about their concrete structure and expression in code (Chapter 6) and their textual description can vary from being over- to underspecified [300]. VanHilst and Fernandez conjecture that the description of security patterns is very abstract, which can lead to many variants in the implementation of the same security pattern [503]. Thus, an obvious challenge is the many implementation variants of a security pattern, which have to be considered.

Due to that fact, it has to be found out how security patterns are shaped in source code to detect them later with a pattern detection tool. Therefore, so-called micro-architectures of security patterns are needed, which can be used for the recognition process. A micro-architecture is a composition of classes, methods, and fields; their structure and organization express an occurrence of a design motif in code [197]. Unfortunately, to the best of our knowledge, no approach exists that deals with the reconstruction of micro-architectures retroactively after a pattern has been described. Thus, this chapter presents a micro-architecture discovery approach and the distilled pattern variants. Since Android apps are highly available (through Google Play [190]) and have often documented security problems [352], we chose to apply our technique to the Android API.

First, this chapter presents well-known pattern mining techniques and illustrates the developed reconstruction process in Section 8.1. Afterwards, security patterns relevant to develop Android applications are selected in Section 8.2. Based on these selected security patterns and the developed reconstruction method, Section 8.3 depicts the located pattern variants. Concluding in Section 8.4 we discuss our observations on the collected variants.
8.1 Discovering Patterns and Micro-Architectures

Security patterns have the issue that they often describe the solution for a problem abstractly without presenting a best practice micro-architecture. Therefore, such micro-architectures have to be detected and documented to be able to search for software patterns in source code.

To the best of our knowledge, the concretion of patterns (if not given in the description) is not clarified or documented. Thus, Section 8.1.1 describes the mining process of patterns in general and Section 8.1.2 introduces the developed process to derive micro-architectures from existing security patterns.

8.1.1 Mining Patterns

Pattern mining is the discovery process of patterns contained in our minds or in activities associated with a certain target [251]. It is a kind of knowledge discovery, but the question arises how to detect such patterns in a mass of data. The discovery process consists of two parts: the way of contribution and the identification of one’s own knowledge.

Way of Contribution

Besides asking the right questions to identify patterns, another question is “who” can discover patterns. Three well-known mining approaches exist:

*Individual contributions* are contributions based on personal expertise or experiences [392].

*Second-hand contributions* are often done through an expert interview or dialogue [252, 392, 523]. Moreover, mining by reading is also a form of second-hand contribution [392].

*Collaborative contributions* are made by a group of individuals mining patterns by having a discussion, e.g., in a workshop, meeting or class [392].

Furthermore, Hanmer and Mirakhorli suggest to use tool support to mine patterns in software [219]. They mention automated design discovery techniques, crowdsourcing, and web mining as suitable practices to help to identify patterns.

Knowledge Identification

Everything starts with an idea. Thus, some works state that an “idea” is needed to identify a pattern [218, 223, 523]. The Hillside Group suggest to answer the following two questions to get such a pattern idea [241]:

- What knowledge would be lost to the company if I were to leave tomorrow?
- What do I know that I have done a thousand times that I think everyone already knows?
A more detailed process has been presented by Fehling et al. [140]. They split the pattern discovery and usage process into three subprocesses: pattern identification, pattern authoring, and pattern application (Figure 8.1a). The pattern identification depicts a method to retrieve a pattern idea (Figure 8.1b). The process starts with the domain definition (domain definition). Afterward, the source of information should be considered (coverage consideration), the collected information homogenize (information format design) and the persistence of knowledge information in a proper way, e.g., table-centric document or web-based tool (information collection). In the last step the collected information is reviewed (information review) to have a smaller set of existing solutions for the pattern authoring process.

After an initial idea has been found, Alexander suggests the steps Identify the subject of pattern, Identify the problem that this pattern resolves, and Identify invariance to substantiate the pattern idea [9].

Iba and Isaku introduce a holistic approach for pattern mining [251]. They propose a structured group mining process based on one’s own experience and visual clustering. The technique requires several rounds of interviews or workshops to solicit the design knowledge from experts with regard to the elements of a pattern and the details. Later on, Iba developed a pattern writing sheet that allows one to collect the required pattern information step by step by filling out their sheet [250].

Pattern mining is also often included in the pattern writing process that starts with describing the pattern’s solution (Figure 8.2) [523]. Besides, the process steps are often described as patterns themselves within a pattern language [218, 223, 329]. Hanmer and Mirakhorli present several patterns that describe pattern mining with regard to software analysis and maintenance purpose and how tools can support the identification [219].
8.1.2 Discovering Micro-Architectures based on Patterns

The introduced knowledge discovery processes are very general to help one to distill patterns out of any data and prioritize different aspects of the mining process. Thus, with regard to security patterns, the described techniques cannot be used one by one in reverse order to discover security patterns in code.

Way of Contribution

Developers can be good partners to detect frequently used mechanisms that are implemented in software systems. Thus, conducting an interview or questionnaire are probably proper tools to do so. Unfortunately, many programmers are not aware of programming security into a software system and often not well-skilled with regard to security [324, 502]. This raises the problem that even by introducing some security patterns, developers are maybe not aware of using them. On account of that they may have problems in understanding the security issue and how appropriate countermeasures could be implemented in code. Possibly, another problem may be that they do not capture the core implementation of a pattern. Concluding, to perform such a process at least one programmer or system architect with software security, design pattern and reengineering skills is needed. Finding programmers or systems architects that have the required skills and time to participate in an interview or workshop is difficult. Thus, we selected two ways of contribution for the presented approach: individual contribution and second-hand contribution by reading related articles in books and the Internet.

Knowledge Identification

The pattern mining process starts with an idea or domain selection. Since developers are often not well-skilled in programming security features [324], the Open Web Application Security Project (OWASP) suggests “the most effective way to bring security capabilities to developers is to have them built into the framework.” [90]. According to this statement and other requests forcing developers to use existing frameworks and libraries to implement security features (e.g., [260, 359]), many
developers will not implement security features by themselves. Rather, they use the security features provided by the framework or library to secure their applications. Moreover, Gamma et al. mentioned in their design patterns book that they observed many patterns by inspecting common frameworks and libraries [173]. They assume that these reusable software pieces provide well-established solutions because of their frequent usage and well-designed architecture. Thus, we “make patterns more specific to a particular domain” as suggested by Yoshioka et al. by analyzing frameworks and libraries [539].

**Procedure**

First of all, the pattern description is read and the security problems to be solved by the pattern is distilled (Figure 8.3). Based on the given information, we looked into Android programming books and searched the Internet for clues implementing the described security feature. Then the collected examples were tested and evaluated as to whether the example code solves the problem described by the selected security pattern. In the next step, essential clues within the API were identified and code examples to extract the necessary parts of using the API. These code examples consist of small code modifications that preserve the security feature of the example and represent their essentials. Based on these minimal reproducers a micro-architecture of the pattern was constructed by identifying structural properties that ensure the security feature on the tested examples.

**Limitations**

Our approach is limited by the chosen framework API and the collected examples of each security pattern. These limitations can be addressed in further work by considering external libraries and further examples expressing security-pattern requirements. For example, sometimes more libraries for the same issue exist. In this case, the software solutions can be compared and similar subsystems or class combinations may indicate a best practice.

### 8.2 Security Patterns and Android Security

As part of the discovering process, a domain has to be selected. In this thesis, the domain of Android applications was chosen. These applications are highly available
(through Google Play [190]) and have often documented security problems [352]. In the following sections, we select patterns that are interesting for the security of Android applications and collect the different variants for each selected pattern.

8.2.1 Patterns Relevant to Securing Android Apps

Since the publication of the GoF design patterns many security patterns have been published (Chapter 3). In that literature review, we identified over 190 security patterns that are related to the Software and Cryptographic application domain (Chapter 4). It is expected that not all surveyed security patterns are relevant to developing secure Android applications.

Guidelines support developers to be aware of important security and privacy issues. They can be used as a kind of checklist before an application is made available. We consider these development guidelines as a good starting point for selecting security patterns that are not strictly based on the context and suitable for Android apps. Thus, the most relevant patterns for Android security are extracted by mapping them to secure developing guidelines.

8.2.2 Collecting Security Guidelines

The development guidelines are collected by conducting an Internet search with the keywords "smartphone app/application" "mobile application" mobile app" combined with "secure", "security", "privacy", "guideline", and “guide”.

<table>
<thead>
<tr>
<th>Guideline by</th>
<th>Main Target</th>
<th>Platform</th>
<th>Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENISA [137]</td>
<td>privacy &amp; security</td>
<td>neutral</td>
<td>English</td>
</tr>
<tr>
<td>ICO [260]</td>
<td>privacy</td>
<td>neutral</td>
<td>English</td>
</tr>
<tr>
<td>Google [188]</td>
<td>security</td>
<td>Android</td>
<td>English</td>
</tr>
<tr>
<td>Düsseldorfer Kreis [36]</td>
<td>privacy</td>
<td>neutral</td>
<td>German</td>
</tr>
<tr>
<td>German IT Summit [258]</td>
<td>privacy &amp; security</td>
<td>neutral</td>
<td>German</td>
</tr>
</tbody>
</table>

Table 8.1 – Collected guidelines discussing secure (Android) application development.

In total, five guidelines dealing with mobile security were found (see Table 8.1). Two of the five guidelines are written in German and the others in English. Three guidelines are issued by governmental or similar institutions [36, 137, 260]. One of the other two guidelines is published by Google Inc. [188], and the last one is issued by a group of companies focused on evaluation and certification in collaboration with the German Federal Office for Information Security (BSI) [258].

Most of the guidelines present the security and privacy issues in purely textual form and independent of a particular software platform. Only one guideline is platform-specific and provides code snippets for the Android platform [188]. The description of each security/privacy issue is often concise and only presents one side of the problem (attack or defense view). Other interesting aspects, such as consequences or a more concrete solution, are rarely given.
8.2.3 Relating Patterns and Guidelines

Potentially relevant security patterns for ensuring security and privacy in mobile apps are determined by reading the collected guidelines. Thereby, the security patterns are aligned to the described security problems. For instance, “you should always use encrypted connections for transmitting usernames, passwords, and particularly sensitive information …” [260] matches with the *Secure Pipe* pattern [464].

All security patterns mentioned in the guidelines were collected. Then the total of the occurrences for each security pattern was calculated. In total, 36 security patterns were mapped to the security issues mentioned in the guidelines. The whole mapping is given in Appendix E. In the next step, the five most frequently mentioned security patterns are inspected.

8.2.4 Selected Security Patterns

The following five patterns are mentioned by all guidelines:

- Information Obscurity [437]
- Secure Channel [437]
- Secure Pipe [464]
- Secure Communication [50]
- Authentication Enforcer [464]

We observed that some security patterns seem to describe the same security issue. The three patterns *Secure Pipe*, *Secure Communication*, and *Secure Channel* depict an encrypted connection. On comparing the three patterns directly with the help of the description section alignment described in Chapter 5, we figured out that they generally describe the same security feature. Therefore, the *Secure Pipe* pattern was chosen as a representative for encrypted communication.

The other patterns in the selected subset are distinct. Finally, the following security patterns for the further detection process were selected:

**Authentication Enforcer** enforces the authentication of a subject.

**Information Obscurity** generates a cryptographic key that can be used for encrypted storage.

**Secure Pipe** establishes a secure communication between two communication parties.
8.3 Security-Pattern Variants

The process mentioned above is time-consuming. Therefore, the focus lies on three patterns that fulfill often needed requirements in software systems. To detect patterns in a software system, code objects that are involved in an implemented security pattern have to be identified and the object interactions have to be specified. This section depicts the detected variants for the previously selected patterns.

8.3.1 Authentication Enforcer

This pattern presents a mechanism for providing identification and authentication from a client to a server to retrieve a remote object [464]. Figure 8.4 shows the four participants of this pattern. The Client uses the AuthenticationEnforcer to authenticate a user. The AuthenticationEnforcer uses the credentials passed in the RequestContext for authentication. The RequestContext contains the (protocol-specific) user credentials and the Subject instance represents the authenticated user.

For the following variants apply that everything is processed within the client application. Thus, a Client participant cannot be located explicitly and the created Subject is often handled internally and cannot be located (except in Variant A8).

![UML class diagram of the Authentication Enforcer pattern](see also [464]).

**Variant A1 and A2**

Authentication Enforcer variant A1 consists of two classes. At first, a URL object is created with Java’s “new” operator and a String or other URL containing the target host for the connection. In the next step, by calling openConnection() a URLConnection object is created. For this connection an authentication can be added by calling the setRequestProperties()-method. This variant is often used with Base64-encoded-username and password information (Figure 8.5a). As URI objects in Java can also create URL objects, the aforementioned authentication variant can also be set up initially with a URI object (variant A2 Figure 8.5b). The classes of both variants combine the participants AuthenticationEnforcer and RequestContext of the security pattern.
Authentication Enforcer & Request Context

Priority Enforcer

Authentication Enforcer & Request Context

Authentication Enforcer

uses

Authenticator

+ Authenticator()
+ setDefault/Authenticator()
+ getPasswordAuthentication():PasswordAuthentication

Request Context

creates

PasswordAuthentication

+ PasswordAuthentication(String,String)

Variant A3

This variant provides the authentication information for a connection request based on URLs. This Authentication Enforcer is—if there is no particular set up on domain names—automatically used for all later created URL connection in the source code. It consists of two objects, to be specific, the Authenticator class as authentication enforcer entity and the PasswordAuthentication class forming the security pattern’s request context. The abstract Authenticator has to be configured with a new instance of Authenticator by calling the setDefault() method. Any Authenticator instance must override the getPasswordAuthentication() method. Thus, a new Object of type PasswordAuthentication with user name and password information has to be created and returned in the getPasswordAuthentication() method. The encapsulated credentials are used for the authentication.
Variant A4 and A5

Since Google integrated some Apache libraries into the Android framework, classes of the Apache HttpComponents library are natively available in the framework [169]. With this library it is possible to create HTTP connections by using a subclass of the AbstractHttpClient class, such as the DefaultHttpClient, as depicted in Figure 8.7. Two ways are possible to enhance the connection with authentication information. On the one hand, an implementing class of the CredentialsProvider interface can be created, for example, by instantiating BasicCredentialsProvider and then plug the instance into the DefaultHttpClient instance by calling the setCredentialsProvider() method (Figure 8.7a). On the other hand, a new DefaultHttpClient instance can be created. Further, the method getCredentialsProvider() is called and returns an instance that implements the CredentialsProvider’s interface (Figure 8.7b). Each concrete CredentialsProvider instance can be configured with user’s credentials by setting up the required information in a new Credentials instance.

![Authentication Enforcer pattern](a)

![Authentication Enforcer pattern](b)

**Figure 8.7** – UML class diagrams of the Authentication Enforcer variants A4 and A5.

Variant A6 and A7

The natively available Apache HttpComponents library in the Android framework can be set up in a similar way to Authentication Enforcer variant A1. The HttpClient uses an HttpMessage and its setHeader() method to set the “Authentication” aspect with two strings on connecting to the HTTP server. The
Android framework provides two implementations for the `HttpClient` interface, namely `DefaultHttpClient` and `AndroidHttpClient` (Figure 8.8a and 8.8b). Both classes can be used to establish an HTTP connection.

![UML class diagrams of the Authentication Enforcer variants A6 and A7.](image)

(a) Authentication Enforcer pattern variant A6.  
(b) Authentication Enforcer pattern variant A7.

**Figure 8.8** – UML class diagrams of the Authentication Enforcer variants A6 and A7.

**Variant A8**

Variant A8 of the Authentication Enforcer depicts a key feature of the Android system (Figure 8.9). Here, the `AuthenticationManager` object uses the relevant `AbstractAccountManager` that creates and provides access to a specific `Account` object for authentication. The `AbstractAccountManager` creates an `IBinder` object that is used by an Android `Service`. The `AuthenticationManager` uses the registered `Service` and its `AbstractAccountManager` object that provides the authentication information as an `Account` object and the authentication mechanism itself. The `AuthenticationManager` is the Authentication Enforcer participant and the `Account` object is the Request Context participant. The Subject participant is self-written and provides the subject information in form of a token. This information is returned, e.g., with calling the `getAuthToken()` method of the `AccountManager` object.
8 – Security Patterns in Source Code

8.3.2 Information Obscurity

This pattern describes data obscurity by using encryption and decryption mechanisms [437]. It prevents other people or systems from reading sensitive information. The pattern has six participants according to its description (Figure 8.10). The Application Component uses the Encryption Mechanism to obtain the Encryption Key to “obscure” the Data. The Key Storage Mechanism stores and possibly distributes an Encryption Key. The Protected Location stores the encryption artifacts used by the system.

The pattern participants Data, Application Component and Protected Location depend on the application design and cannot be mapped explicitly, which applies for all following Information Obscurity variants. For example, the participant Protected Location, which hides the Encryption Key, is not explicitly necessary and is often not used due to the usage of password-based encryption where the password is entered directly.

![Figure 8.10 – UML class diagram of the Information Obscurity pattern (see also [437]).](image)

**Variant I1**

The first extracted variant for the Information Obscurity pattern is shown in Figure 8.11. For this variant, the Encryption Key and Encryption Mechanism are essential participants.

The Encryption Key participant consists of a SecretKeyFactory object. This factory must be initialized with getInstance() and the specific key generation algorithm. To generate a SecretKey object by calling SecretKeyFactory’s method generateSecret() an implementing class ofKeySpec is needed to initialize the key.
material, e.g., a PBEKeySpec object to manage password-based key generation (see also Listing 9.1). A previously created Cipher object can then use the generated SecretKey by calling the method init() and encrypt or decrypt data by calling doFinal().

![Figure 8.11 – UML class diagram of the Information Obscurity variant I1.](image)

**Variant I2, I3, I4, and I5**

The variants I2 to I5 use a KeyGenerator object that creates an object implementing Key. This factory must be initialized with getInstance() and the specific key generation algorithm. The variants differ in the following configuration options of the KeyGenerator:

- with an integer value (Figure 8.12)
- with a SecureRandom object (Figure 8.13)
- with a AlgorithmParameterSpec object (Figure 8.14)
- with a AlgorithmParameterSpec and a SecureRandom object (Figure 8.15)

by using the init() method. Analogously to the already described variant I1, a previously created Cipher object can use the created Key object by the KeyGenerator.

![Figure 8.12 – UML class diagram of the Information Obscurity variant I2.](image)
Variant I6

For this variant a CertificateFactory has to be created by invoking the getInstance()-method. The Certificate is created by calling on the CertificateFactory object (Figure 8.16) the createCertificate() method. The created Certificate can now be used by a Cipher object's init() method to encrypt or decrypt data.
8.3 – Security-Pattern Variants

**Figure 8.16** – UML class diagram of the *Information Obscurity* variant I6.

**Variant I7 and I8**

The variants I7 and I8 use both a KeyFactory object initiated with a `getInstance()` method. Based on a given KeySpec object, a private (Figure 8.17) or a public (Figure 8.18) key is created. That key can be used to set up the Cipher object for encrypting or decrypting given data.

**Figure 8.17** – UML class diagram of the *Information Obscurity* variant I7.

### 8.3.3 Secure Pipe

This pattern establishes a secure connection between two systems. In the description by Steel et al. [464], the pattern consists of the participants *client*, *secure pipe* and *application* (also known as server) (Figure 8.19). Their interaction is described as follows: The *client* sends a request to the *server* and the *server* creates a *secure pipe*. The *secure pipe* negotiates parameters of the connection with the *client*. The
client uses the secure pipe to encrypt a request to the server, which in turn decrypts the request. The server can also establish a secure pipe with the client. Technically, this pattern can be split into a server and client side for establishing the secure connection (e.g., Variant S1 and S2 (Figure 8.20)).

Variant S1, S2, S3, and S4

The micro-architectures for creating SSL sockets on server and client side are shown in Figure 8.20. An essential object of this pattern is the SSLSocketFactory, which inherits from SocketFactory. By calling the getDefault() method an SSL-configured factory object is created, which in turn creates an SSLSocket object with the createSocket() method call. The counterparts on the server side and the related objects are SSLServerSocketFactory and SSLSocket, respectively.

Another way to create a SSLSocketFactory or SSLServerSocketFactory object is to use the SSLContext object of the Android API. A default object is returned by calling the static getDefault() method of the object. The returned object can now be used to obtain a SSLSocketFactory/SSLSocketFactory
8.3 – Security-Pattern Variants

(a) Creates a Secure Pipe on client side (Variant S1).

(b) Creates a Secure Pipe on server side (Variant S2).

Figure 8.20 – Variant S1 and S2 for creating a Secure Pipe.

(a) Creates a Secure Pipe on client side (Variant S3).

(b) Creates a Secure Pipe on server side (Variant S4).

Figure 8.21 – Variant S3 and S4 for creating a Secure Pipe.

object by calling the `getSocketFactory()`/`getServerSocketFactory()` method. As mentioned before, a `Socket/ServerSocket` can now be created by using the associated factory object.

Variant S5

As mentioned before, it is also possible to use classes of the Apache library V4.0 which have been integrated into the framework at its first release. Nowadays, the classes are depreciated with API level 22, but can still be used for developing new apps. Thus, objects and derived objects of `org.apache.http.conn.ssl.SSLSocketFactory` can be used similarly to the Java API `SSLSocketFactory` to create secure sockets (Figure 8.22).

Variant S6, S7, S8, and S9

Another often used possibility to communicate is to use HTTP. This protocol is often used in Internet communications, e.g., with web servers. This communication can be secured by using HTTPS instead of HTTP. HTTPS adds the capabilities of SSL/TLS to the HTTP communication. Figure 8.23 depicts two variants of opening a URL connection that can be secured with SSL. The given parameter during the URL object creation defines the connection target. If the URL name starts with HTTPS
the Android/Java runtime automatically secures the connection with SSL/TLS, e.g., in Figure 8.23a an `HttpsURLConnection` instead of `HttpURLConnection` is created. A URL object can create a subtype of `URLConnection` by calling the `openConnection()` method (Figure 8.23a) or a subtype of `InputStream` by calling the `openStream()` method (Figure 8.23b).

![Figure 8.22 – UML class diagram of the Secure Pipe variant S5.](image)

Figure 8.23 – UML class diagrams of the Secure Pipe variants S6 and S7.

The aforementioned two variants based on `URL` can also be created by using `URI` objects (Figure 8.24). `URL` objects can also be created with `URI` objects. In such a case the `URI` instantiation statement contains the connection information that is needed for enabling security features for the connection.

**Variant S10 and S11**

The figures 8.23 and 8.24 show two variants establishing HTTP connections using the normal Java API. Another possibility to create HTTP connections by using the in the Android framework integrated Apache HttpComponents library [170] is depicted in Figure 8.25. One obvious difference is the amount of classes that can be used to establish a connection. Another difference is the use relation between `HttpClient`
8.3 – Security-Pattern Variants

(a) Secure Pipe pattern variant S8.
(b) SecurePipe pattern variant S9.

Figure 8.24 – UML class diagrams of the Secure Pipe variants S8 and S9.

and HttpRequest which differs from the creational relationship between the objects in the variants S6, S7, S8, and S9. Moreover, based on differentiated HttpRequest objects the HTTP request can be programmed in a diverse way.

Figure 8.25 – UML class diagrams of the Secure Pipe variants S10 and S11.
Table 8.2 – Number of extracted security pattern variants.

<table>
<thead>
<tr>
<th>Security Pattern</th>
<th>Variants</th>
</tr>
</thead>
<tbody>
<tr>
<td>Authentication Enforcer [464]</td>
<td>8</td>
</tr>
<tr>
<td>Information Obscurity [437]</td>
<td>8</td>
</tr>
<tr>
<td>Secure Pipe [464]</td>
<td>11</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>27</strong></td>
</tr>
</tbody>
</table>

8.3.4 Result

Table 8.2 shows the number of detected variants per security pattern for the Android framework. Eight different variants have been considered that implement an Authentication Enforcer pattern. The variants combine up to two or three different objects to implement the pattern. The Information Obscurity pattern can be implemented in eight different ways. The variants differ from two interacting objects to four, but mostly in the kind of the used encryption key, e.g., private/public key-based, certificate-based or password-based. The Secure Pipe pattern can be implemented by using the Android API in eleven different ways. The detected variants consist of two or three interacting objects and one or two connections to provide the described security feature.

We found several variants for security patterns. All these variants may additionally differ with regard to classes implementing interfaces or inheriting from (abstract) classes to specialize the security feature expressed by the security pattern. One example are the aforementioned Information Obscurity variants (see Section 8.3.2), where 14 subinterfaces of the interface Key can be used to set up the different types of encryption. For example, while focusing on the implementation, Secure Pipe’s variant S7 consists of two classes to establish an HTTP connection. Its variant S11 requires also two objects but can be configured with many objects inheriting the HttpRequest class to specify exactly which type of request is needed, e.g., an HttpGet or HttpDelete request. Moreover, the variants per pattern can vary based on the context, e.g., the transport media. An example are the Secure Pipe pattern variants where socket or HTTP-based communication is secured. All in all, the collected variants are manifold, and each variant addresses specific security configurations.

8.4 Summary

In this chapter, we have illustrated how patterns can be extracted from an API. First, we have collected security patterns that can be used to secure Android applications by mapping them to secure coding guidelines. Second, we have developed a procedure to extract coding variants of security patterns of an API and have shown that security patterns are implemented into the Android framework.

Based on the collected examples and the variants, we can say that implementing security features in source code may be quite diverse. More or less configurable
objects are used to provide specific features. Thereby, we can confirm the assumption made by VanHilst and Fernandez that security patterns show more variability in details [503], for example, the Information Obscurity pattern variants I2, I3, I4, and I5 which differ basically in the input objects for the key generation. They also consist of several interacting objects and several variants exist for one pattern (e.g., the Secure Pipe pattern can be implemented in eleven different variants). The implementations variants are highly customizable (e.g., Secure Pipe variant S10 and S11) and some objects provide security checks within their API implementation to ensure security (e.g., Secure Pipe Variant S6).

All in all, the security pattern implementations are more loosely based on security requirements than pattern dictated specific implementation details or programming language restrictions as we can see by the GoF pattern variations [513]. Nevertheless, their several interacting objects can be mapped to security pattern participants which can give relevant security implementations in the code a comprehensible name. Thereby, it can help the analyst to deal with closely related security functions, which are also relevant or influence a particular functionality.

The security properties derive from a correct application of a pattern’s objects, and the implementation verification requires to find the corresponding piece of code in a software system [503]. Therefore, a security pattern detection tool should find interacting objects that express a specific security pattern to support analysts in assessing a software system. Moreover, new technologies, designs, and requirements will arise over time which will reside in new security patterns variants. Hence, a pattern detection tool should be configurable and extensible. For example, in third-party libraries or tools, which are used or new security requirements, require newly implemented features, or modified objects to satisfy the security demand. In the following chapter, a tool is presented that considers this aspect and provides low-level information of detected security patterns to take the results of this chapter into account.
CHAPTER NINE

SECURITY-PATTERN DETECTION TOOL

Static analysis tool for Android applications such as FlowDroid [26], Fortify [168], or AppScan [254] aim to detect vulnerabilities or (sometimes) tainted flows and present the reviewer detected possible issues of an analyzed Android application. However, none of these tools supports the identification of implemented security features in code, although this aspect is also relevant to developers as well as reviewers.

To address this open problem, this chapter presents an approach based on Object Process Graphs (OPGs). It enhances security program comprehension by detecting interacting objects in source code described by security patterns and finally visualizing them.

9.1 Introduction

We have seen in Chapter 8 that interaction points between (Java) objects implementing security patterns may influence their security setup or functionality. A way to represent an object configuration is to extract an Object Process Graph (OPG) for a single object [376]. Such a graph describes the overall control flow of a single object within an application. An OPG can be used for protocol validation purposes [375] as well as program comprehension tasks [373]. The existing OPG approaches, however, are limited to only one object in code. Thus, an extension of this approach is needed to track multiple objects and their interaction. Assisted by an object configuration visualization, an analyst can answer question Q6 (see Chapter 1 and Figure 9.1) more easily by having a security pattern’s general goal and the reviewed implementation in mind. Moreover, a given software architecture documentation can be validated and possibly completed by adding detected security patterns as suggested in Chapter 7.

This chapter is structured as follows. Section 9.2 motivates the detection and visualization of multiple-object interaction for analysts with the help of an example based on a security-pattern, followed by Section 9.3.2 introducing the original OPG concepts and our extensions to shape COPGs. Section 9.4 presents the analysis tool,
Q1: Where are security feature locations?
Q2: What should this security feature do?
Q3: Which (Java) code objects shape this feature?
Q4: Where do these (Java) objects interact in code?
Q5: Is the interacting object configuration correct or faulty?
Q6: Does the implemented security feature meet the requirements and architecture description?

Figure 9.1 – An analyst’s challenges on assessing an implemented security feature (see also Section 1.1).

which uses COPGs to detect multiple interacting objects and Section 9.5 provides a short summary.

9.2 Motivation

Chapter 1 already illustrated the challenges an analyst has to face on assessing security features in the code. For example reconsider question Q3 of Figure 9.1. Here, the analyst must know which combinations of (Java) objects form a specific security feature. Therefore, the analyst must have good knowledge of security mechanisms and features in general, but also much experience on how they must be implemented in code.

Let us consider a more concrete situation where the implementation of a data encryption functionality is to be manually reviewed, e.g., during maintenance tasks when a vulnerability should be fixed or due to quality checks before the software is going to be released.

The listings 9.1 and 9.2 illustrate an encryption example to be assessed. During the review process, an analyst must locate the code where the encryption is done. In particular, she must check that an encryption algorithm of adequate strength is employed (Listing 9.1 line 6). She must also assure that the intended data are encrypted (Listing 9.1 line 8), which may require to trace the data flow back to its source (which can be located in a different software module). Similarly, the encryption setup and the key generation process must be clearly understood (Listing 9.1 line 5 and Listing 9.2 lines 2 to 13). Again, an analyst must find the corresponding code locations to rule out weaknesses, such as usage of weak encryption algorithms (Listing 9.1 line 6), hard-coded secrets (Listing 9.2 line 3) or faulty seeding of a random number generator. The analyst must be aware of the interaction between the Cipher and the SecretKey object, which provides the complete security functionality for data encryption.

For (security-skilled) maintenance programmers and security experts it is a time-consuming and challenging task to separate correct from faulty security
9.2 – Motivation

configurations and verify the given application security requirements. The required level of understanding can be supported by using software comprehension tools that are tailored towards security as suggested by McGraw and others [31, 324].

```java
1 public final class Crypto {
2     public final byte[] encrypt(final byte[] plain, final String password) {
3         byte[] encrypted = null;
4         try {
5             SecretKey key = Util.generateKey(password);
6             Cipher cipher = new NullCipher();
7             cipher.init(Cipher.ENCRYPT_MODE, key);
8             encrypted = cipher.doFinal(plain);
9             System.out.println("data encrypted");
10         } catch (Exception e) {...}
11         return encrypted;
12     }
13 }
```

**Listing 9.1** – Encryption example.

```java
1 public final class Util {
2     public static final SecretKey generateKey(String password) {
3         password = "myDefaultTestPassword";
4         byte[] salt = new byte[20];
5         SecureRandom sr = new SecureRandom();
6         sr.nextBytes(password.getBytes("UTF-8"));
7         SecretKey secKey = null;
8         try {
9             SecretKeyFactory keyFactory = SecretKeyFactory.getInstance("PBEWITHSHAAND128BITAES-CBC-BC");
10             PBEKeySpec keySpec = new PBEKeySpec(password.toCharArray(), salt, 1024, 128);
11             secKey = keyFactory.generateSecret(keySpec);
12         } catch (Exception e) {...}
13         return secKey;
14 }
```

**Listing 9.2** – Encryption example help file.

Based on the example above we see the following four steps where analysts can be supported:

1. **Detecting source code locations implementing security features**

   requires existing security skills in combination with knowledge about the software frameworks and libraries used in an application [120] (question Q1 and Q2 in Figure 9.1). This task can be supported by detecting low-level implementation bugs with tools such as Fortify SCA or IBM AppScan, but it still needs manual effort for further inspection [32, 294]. Furthermore, (partly) correctly implemented security patterns cannot be found with such tools.

2. **Tracing the configuration of an object**

   by looking at source code in an editor or IDE can be an easy task when the object is locally defined and used. However, tracing an object that implements the concept of interest might be very difficult when its code is scattered across multiple components or tangled with code implementing other concepts. An analyst must remember relevant code locations or note them down somewhere else. Furthermore, she must jump through the source code to backtrack security-relevant Java objects (and in
some cases jump back and forth). In principle, an analyst must construct a mental trace of the objects to be analyzed. This process is time-consuming and tedious. Furthermore, one can easily overlook important code locations. In case of dead code—Android apps often use several libraries—the results may be misleading.

3 **Understanding object interactions** is a further task relevant to security code reviews. Security features are often integrated into libraries and frameworks. Due to their adaptability and configurability they are often composed of several objects. Even if an analyst finds an object that implements a security feature and traces the object, she needs to know which other objects (participants) are additionally necessary to implement the feature and ensure a specific state of security (question Q3 and Q4 in Figure 9.1). In our example, the `SecretKey` that is used for the encryption has also to be inspected (Listing 9.1 line 5 and Listing 9.2).

4 **Configuration assessment** can be realized by an analyst when information on the function of code objects, their interaction, the execution context, and the software requirements are consolidated into the analyst’s mental software model. With this information, the overall configuration of the feature can finally be checked and assessed to be correct or faulty (question Q5 and Q6 in Figure 9.1).

A way to represent an object configuration is to extract an object process graph (OPG) for a single object [376]. Such a graph describes the overall control flow of a single object within an application. An OPG can be used for protocol validation purposes [375] as well as program comprehension tasks [373]. The existing OPG approaches, however, are limited to only one object in code. Thus, an extension of this approach is needed to track multiple objects and their interaction. Assisted by an object configuration visualization, an analyst can answer question Q6 by having a security pattern’s general goal and the reviewed implementation in mind. Moreover, the software architecture documentation can be validated and possibly completed by adding detected security patterns as suggested in Chapter 7.

Later in this chapter, a tool is introduced that allows us to automatically extract graphs that contain connected subgraphs representing different interacting objects from source code. This graph can be used to support the steps 1 and 2 by finding objects of interest in code and displaying their interaction.

### 9.3 OPG Concepts and Enhancements for COPGs

When it comes to understand all (statically) possible sequences of operations on a particular object, object process graphs (OPGs) are regarded as useful [374, 376]. Such OPGs “describe the set of [static] traces relative to a statically detectable object” [124], which allows an analyst to focus on the object’s specific configuration. According to Quante an OPG is a representation that provides an understanding of
9.3 – OPG Concepts and Enhancements for COPGs

Figure 9.2 – ICFG-based OPG construction example for the code from Listing 9.1. Figure (a) shows the ICFG, where the grey boxes indicate atomic nodes of the Cipher object. The crossed parts of the graph are removed to derive the OPG depicted in Figure (b).

how a component is being used within an application [374]. Hence, OPGs may also support a security analyst within the task of software security comprehension when inspecting objects that implement security features.

Unfortunately, the security pattern variants of Chapter 8 are very distinct and cannot be easily transformed into an abstract micro-architecture that would allow generalized detection steps for security patterns. In combination with the aim to enhance security program comprehension, we decided to advance the aforementioned OPG technique to consider object interaction. First, the general concepts of an OPG are shown and then our enhancements on required nodes and the extraction process to generate a Connected Object Process Graph (COPG) are presented.

9.3.1 Object Process Graph (OPG)

An OPG represents the behavior of an individual object extracted from a program. Local or global variables as well as the allocation of new objects (e.g., in Java by way of the new operator) are statically detectable objects. An OPG is an excerpt of an inter-procedural control flow graph that contains only those parts that are relevant for the given object. With an OPG, for example, one can track a Cipher object throughout its lifetime and understand where and how it is initialized as well as under which conditions and where it is accessed (e.g., adding data to be encrypted).

Nodes and Edges

Each node in the OPG represents a location in the program. OPGs typically consist of the following node types:
• \textit{start}—indicates the entry point of the OPG.

• \textit{create}—denotes the creation of an object by way of variable declaration of allocation.

• \textit{read} and \textit{write}—represent access of an object’s attribute.

• \textit{decision}—models a point where the control flow can take two different paths, depending on a boolean value that has been calculated in the previous operation or call.

• \textit{call}—models the interprocedural control flow.

• \textit{atomic call}—is a call to an operation that belongs to the interface of the regarded component.

• \textit{entry}—marks the entry of a called method.

• \textit{return}—marks leaving a method and leads back to the corresponding call node.

• \textit{exit}—indicates the end of object life in the program.

Besides the nodes, edges represent the control flow between the nodes. They can be unconditional (\textit{call}, \textit{return} and control-flow edge) or conditional (true, false) after a \textit{decision} node.

\textbf{Extraction}

OPGs can be extracted statically or dynamically. While dynamic OPG extraction requires program instrumentation for data collection [376], its static counterpart requires interprocedural control and dataflow analysis [126]. OPGs can be statically extracted with techniques similar to (forward) program slicing [519]. To derive an OPG statically, one needs to follow the data flow for a given object from where it is allocated to the points where it ceases to exist. Along these control paths, one only keeps the operations referring to this object (called \textit{atomic} node) and the conditions upon which these are control dependent. Figure 9.2 shows how to construct an OPG from an inter-procedural control-flow graph (ICFG). Figure 9.2a depicts the ICFG of the source code example shown in Listing 9.1 and 9.2 (see Section 9.2). The grey boxes indicate atomic nodes of the \texttt{Cipher} object. The crossed parts of the graph are removed to derive the OPG depicted in Figure 9.2b. The statements marked with the grey boxes shown in Figure 9.2a are now separated from other statements not related to the \texttt{Cipher} object and “labeled” according to their function on the object. For example, the create node is mapped to the Cipher creation statement \texttt{Cipher cipher = new NullCipher();} and the following call node is mapped to the \texttt{init()} method call on the \texttt{Cipher} object.

\textbf{9.3.2 Connected Object Process Graph (COPG)}

The base of our approach is formed by OPGs. Addressing the need for interacting objects, further nodes were added, and the input and extraction process was enhanced.
Nodes and Input

The following nodes were added to the aforementioned OPG node types:

+ **connection**—mark the initial interaction between two objects.

+ **shared**—indicate further interaction between two objects.

OPGs capture the trace of one object that is initialized with the "new" statement in source code only. However, some objects are created from other objects (e.g., factories), which can happen deep inside the API implementation. Such newly-created objects cannot be reliably found and traced by static analysis. Besides, we are only interested in the point of existence in application code. Thus, two new starting points were added to capture the static and sequential creation of objects.

+ **CreatedByStaticObject** node—marks a starting point where an object is created inside another object and is returned by calling a static method, e.g., Listing 9.2 line 9:

  ```java
  SecretKeyFactory keyFactory = SecretKeyFactory.getInstance(...) ;
  ```

  It uses three arguments to define the starting point:

  1. Type name of the static object, e.g., *SecretKeyFactory*.
  2. Method name that returns the created object, e.g., *getInstance*.
  3. Type name of the created object, e.g., *SecretKeyFactory*. Due to different method signatures of methods having the same name, this argument is needed to identify the return type of interest.

+ **CreatedByPreviousObject** node—describes a starting point where an object is created by a previously detected object and is returned by calling a specific method, e.g.,

  ```java
  SecretKey secKey = keyFactory.generateSecret(...) ;
  ```

  using a priorly created *keyFactory* object to generate a key (Listing 9.2 line 11).

Besides, to model the interaction between the object starting points, we defined:

+ **Use** edge—represents the fact that one object uses another one. For example, the *Cipher* object uses a *SecretKey* object for its initialization (Listing 9.1 line 7)

+ **Create** chain—models two or more objects that create each other in sequential order.
Listing 9.3 – COPG input example.

An example of defining a COPG input, which captures a variant of the Information Obscurity pattern, is shown in Listing 9.3. It is based on the code example presented in Section 9.2. The input definition has two starting points: first, the SecretKeyFactory, which is created by a static object in code (Listing 9.3 line 1); second, any object inheriting from the Cipher class that is created with the “new” operator (Listing 9.3 line 3). The SecretKeyFactory object creates a SecretKey object (Listing 9.3 line 2 and 5) that is used by the Cipher object (Listing 9.3 line 6). Finally, all create and use relations are added to the COPG analysis (Listing 9.3 line 9 and 10).

As it is common and often necessary to extend classes to implement security features, we added the feature of using object hierarchies downwards (transitively) for finding objects of interest. By using this function, an interface or superclass is given as input, and the class hierarchy tree is considered for generating OPGs and their interactions. Moreover, connections between objects are automatically detected by traversing the methods of the objects. Thus, API changes, such as method renaming, require no change in the input definition of objects, i.e., the analysis process remains stable in case of API changes. Analogously to the depicted example, our approach can be easily extended for self-defined interacting objects by defining the starting points and their interactions.

Configurability

As the COPG input feature shown in Listing 9.3 supports basic COPG concepts (e.g., CreatedByPreviousObject and CreatedWithNew), one can design a COPG definition language from it. The aforementioned programmatic way of describing the COPG input, however, hampers the distribution and self-development of such connected objects of interest. Thus, we defined an XML-based input format that captures the starting points and relations above (Listing 9.4). Furthermore, a graphical user interface to create the XML input is available (Figure 9.3).
We demonstrate the OPG extraction based on the aforementioned input example. For each object of interest, the creation site in code is determined. Based on these allocation sites OPGs are created (Figure 9.4a). In the next step, the OPGs are inspected with regard to statements that are used by connected objects (defined by use or create relation). According to the control flow the first statement that is shared by both OPGs is the connection node (Figure 9.4a grey marked nodes), and possible further shared statements are marked as shared nodes. These matchings are processed for all defined edges declared in the analysis. Finally, all single OPGs that belong to a detected instance are merged into one OPG by matching entry and end nodes of methods and nodes representing statements, such as call nodes, considering their control flow order.
In summary, our analysis technique comprises two analysis steps. First, the OPGs are extracted by a data and control flow analysis comparable to inter-procedural forward slicing. Second, the connecting statements are identified by pairwise comparisons of the OPGs’ statements.

![Diagram of OPGs and COPG](image)

**Figure 9.4 –** COPG construction example based on OPGs. The [l. x] notation refers to the statement line in listings 9.1/9.2. Figure (a) shows extracted OPGs for the Cipher and the SecretKey object. The grey-marked boxes indicate a connection of the two graphs, which form a single connection node in the final COPG (Figure (b)).

### Limitations
Comparing OPGs to find connection statements can become time-consuming. To avoid missing connections between two OPGs, our approach uses a one-by-one statement comparison following the program statement flow to identify the first shared statement and mark the following shared statements as shared nodes. Thus, the number of comparisons to generate a COPG from two objects is quadratic. The more interacting objects are to be considered, the more memory and time are needed to create COPGs. Since we focused on the identified pattern variants in Chapter 8, we did not check this behavior in depth. We consider this as future work.

### 9.4 Analysis Tool
Our tool is based on the Soot analysis framework [500]. Soot provides several basic static analyses, such as points-to-analyses and callgraph generation, and can be extended by implementing new analyses called “transformers” [501]. Moreover, it supports the input of DEX files as provided by Android applications as well as
9.4 – Analysis Tool

Figure 9.5 – Architecture of our analysis tool.

Java byte and source code. Figure 9.5 depicts the overall tool composition and our analysis enhancements to Soot (step 1 and 3).

9.4.1 Preprocessing

On analyzing one Android app, we must deal with some special issues due to the Android platform design.

**Specific Characteristics of the Android Framework**

The following Android framework characteristics influence the static analysis of Android apps [306]:

![Component lifecycle modeling](image)

**Component lifecycle modeling** In contrast to common Java programs, Android apps are not stand-alone applications and consist of components, such as activities and services, which have a distinct lifecycle (Figure 9.6). The
corresponding lifecycle methods (i.e., `onCreate()`, `onPause`, `onStart()`) are triggered indirectly by the Android framework to handle the app’s execution [183]. This is essential for creating cohesive control-flow graphs which are often needed by further static analyses.

**Program entry points** Android apps consist of several activities and other components. They have no single `main()` method which indicates the program starting point. However, some tools like Soot need a "main"-method as input, so an artificial main method considering the components of an app has to be created.

**Inter-Component Communication (ICC)** The components of an Android app (i.e., activities and services) can interact via *Intents* [185]. This interaction has to be modeled to connect the control-flow graph of single components to a system dependence graph (SDG). This enables further analyses to work across components.

```java
public class MyActivity extends Activity{
   ...
   Intent serviceIntent = new Intent(this, MyService.class);
   startService(serviceIntent);
   Intent intent = new Intent(this, SecondActivity.class);
   startActivity(intent);
   ...
}
public class SecondActivity extends Activity{
   public void onCreate(...) {} 
}
public class MyService extends Service{
   public void onCreate(...) {} 
}
```

*Listing 9.5 – Android ICC example.*

**User Interface (UI) callbacks** The Android framework provides specific UI callbacks, e.g., to react on a user’s button click via `onClick()`-Listener or system events such as `onCellStateChanged()` [26, 356]. Similar to the lifecycle methods (Figure 9.6), these callbacks are called by the framework. Modeling these actions enhances also a created control-flow graph for further analyses.

```java
protected void onCreate(...) {
   Button button = (Button) findViewById(R.id.myButton);
   button.setOnClickListener(new View.OnClickListener() {
      @Override
      public void onClick(View v) {
         handleButtonClick(v);
      }
   };
}
public void handleButtonClick(...) {
   ...
}
```

*Listing 9.6 – Android UI callback example.*

**Handler** Android also provides a specific message mechanism for communicating through threads within an app. This mechanism is called `Handler` and enables threads to communicate through a shared `Handler` object.
AsyncTask Another Android extension to multi-threading is called AsyncTask. This extension, for example, provides a way to communicate with a user interface thread in an asynchronous fashion. Handler and AsyncTask are triggered by calling specific methods through the Android framework, e.g., when on an AsyncTask object the `execute()` method is called the framework invokes implicitly four methods of that object [184]. Listing 9.7 shows an example of the AsyncTask usage. The `onPreExecute()` method is invoked to prepare items like a progress bar. The method `onPreExecute()` in line seven is implicitly called through the method call `execute()` on the DownloadTask object in line 3. Then, the `doInBackground()` method of that object is called to perform the background computation. Thereby the `publishProgress()` method can be used to trigger the method `onProgressUpdate()` to display the computation progress. Finally, the method `onPostExecute()` is called to pass back the computation result.

```java
public class MyActivity extends Activity {
    public void onCreate(...) {
        new DownloadTask().execute(url1, url2, url3);
    }
}

class DownloadTask extends AsyncTask<URL, Integer, Long> {
    protected void onPreExecute(){ ... }
    protected Long doInBackground(URL... urls) {
        publishProcess(...)
    }
    protected void onProgressUpdate(Integer... progress) { ... }
    protected void onPostExecute(Long result) { ... }
}
```

Listing 9.7 – Android AsyncTask example.

XML-Layout The user interface of Android apps is structured by layouts. These layouts can be declared via XML configuration files or Java code. The XML configuration can define UI elements which are used via string reference within Android components. The Android framework instantiates the defined UI elements itself. These defined elements have to be considered and modeled to provide a more complete analysis scenario for an app.

```xml
<Button android:id="@+id/myButton"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="My Button"
    android:onClick="handleButtonClick"/>
```

Listing 9.8 – Android XML layout example.

```java
protected void onCreate(...) {
    Button button = (Button) findViewById(R.id.myButton);
    public void handleButtonClick(...) { ... }
}
```

Listing 9.9 – Android layout example.
The basic Soot tool does not provide such a simulation of Android-app behavior to be able to generate viable callgraph and pointer information [26]. Some tools simulate the Android framework behavior to enable static analyses for special purposes such as tainted flow or privacy leak analyses [26, 305, 356]. These tools, however, cannot be easily adapted to other analyses. Their modifications to simulate the Android framework are firmly woven into the analysis process and linked to the analysis goal, e.g., to find tainted flows. Thus, it is difficult or even impossible to reuse the Android framework simulation extensions and append new analyses (transformers) to the tool.

Our Approach

Due to the reasons above, we developed our own pre-analysis steps to simulate the Android app behavior, which is normally performed by the Android system, based on the work by Arzt et al. [26]. First, we collect all Android components (e.g., activities, services) that belong to the app. Then, we extract user interface events, which are declared in specific configuration XML files describing user interfaces, and their corresponding callbacks in source code (Figure 9.5, step 1). Based on this information we generate for each Android component a call sequence for user interface interaction and other system events. This step can be compared with the iterative approach by Arzt et al. [26]. The generated call sequences are integrated into a main method, which simulates the overall app lifecycle triggered by the Android framework. Also, we simulate the lifecycle of fragments, which represent a part of the user interface in an activity. In addition, our tool considers and simulates the following aspects:

Application Configuration Android provides the possibility to maintaining the global application state by subclassing the Application class [182]. This class is instantiated before any other activity or service of an application. It is started by the framework on calling the onCreate() method and can be used by services and activities through the getApplication() method which returns the Application object.

Fragment Lifecycle Similar to activities, fragments have a lifecycle with specific methods which are called by the Android Framework (Figure 9.7). A fragment declared in XML is bound statically to the activity that inflated the view which contains the fragment (Listings 9.10, 9.11, and 9.12). This often happens in the onCreate() method of an activity and the onAttach(), onCreate(), and onCreateView() methods of the fragment are called. After the activity’s onCreate() is finished, the fragment’s onActivityCreated() method is called. Then, specific activity methods trigger the lifecycle methods of the fragment (Figure 9.7). The Android framework controls all these actions.

Fragments can also be added, replaced, and removed during a running activity. Therefore, a placeholder within the layout XML is declared by using the <FrameLayout> tag (Listing 9.13). The explicit adding, replacing, and
removing actions through the `FragmentTransaction` object can be found and used to trigger relevant fragment methods (Listing 9.14). Problematic are the calls on a removed fragment when another fragment replaces it and when fragments are moved to a so-called backstack and being reused in the running activity. Due to its indirect addressing through the placeholder it is difficult to determine the actual object by using static analysis. Thus, we do not support these two scenarios.
public MyActivity extends FragmentActivity{
    public void onCreate(Bundle savedInstanceState) {
        setContentView(R.layout.main_layout);
    }
}

Listing 9.10 – Embedding a Fragment statically by inflating an XML layout file.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    ...>
    <fragment
        android:name="com.example.android.MyFragment"
        android:id="@+id/myFragment"
        .../>
</LinearLayout>
```

Listing 9.11 – XML layout file using a fragment.

```
public class MyFragment extends Fragment {
    @Override
    public void onAttach(...) { ... }
}
```

Listing 9.12 – A Fragment example class.

We currently support programmed Fragments controlled via the TransactionManager and its FragmentTransaction object (Figure 9.7) as well as the calling of fragment methods bound to an activity. This allows us to model the usage of fragments within activities partially and thereby to enhance the callgraph and point-to analysis.

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    ...>
    <FrameLayout
        android:id="@+id/my_placeholder"
        android:layout_width="match_parent"
        android:layout_height="match_parent">
    </FrameLayout>
</LinearLayout>
```

Listing 9.13 – XML layout file using a placeholder for dynamically processed Fragments.

```
FragmentManager fragmentManager = getSupportFragmentManager();
FragmentTransaction fragmentTransaction = fragmentManager.beginTransaction();
MyFragment fragment = new MyFragment();
fragmentTransaction.add(R.id.my_placeholder, fragment); // adds the fragment
fragmentTransaction.commit(); // executes the transaction
fragmentTransaction.replace(R.id.my_placeholder, new MySecondFragment()); // replaces the current fragment in the placeholder location with a new fragment
fragmentTransaction.commit(); // executes the transaction
```

Listing 9.14 – An example for adding a Fragment dynamically.
Use Simulator Extracted API We extracted the *android.jar* files of running Android simulators for API level 3 to 18 to have fully implemented API information instead of stubs, which are usually included in the *android.jar* distributed by the Android SDK. The extracted *android.jar* files enhance the generated points-to-analysis information for used API objects.

Java Concurrency Concepts We also simulate framework features related to Java concurrency concepts, such as *Runnable*, *Thread*, *TimerTask*, *ForkJoin*, *Callable*, and *ExecutorService*. For example, the missing simulation of *Runnable* objects within the Soot tool are also added by calling a *Runnable* object’s “run” method after locating its “start” method [460] (Listing 9.15). This information enhances the constructed callgraph and points-to information.

```
new Runnable(new CustomRunnable()).start();
```

*Listing 9.15 – Runnable example*

9.4.2 Normal Soot Analyses

Based on these preparing steps our tool uses the available Soot analyses, such as callgraph generation and pointer analyses, to obtain the required information for the pattern detection based on object process graphs (Figure 9.5).

9.4.3 Identification of Security Patterns

The pattern identification analysis is based on the developed COPG generation, which in turn uses created OPGs as described in Section 9.3.2.

Currently Supported Security-Pattern Variants

We have determined in Chapter 8 which security patterns are relevant to Android app development and which interacting API objects shape selected security patterns in application code. These variants are the initial set of security-relevant interacting objects for testing the COPG approach.

Unfortunately, some of the variants have to be excluded from the tool test due to the limitations of our approach. For the variants one and two the method call `setRequestProperty()` and for variant six and seven the method call `setHeader()` is essential to mark the object combination as an Authentication Enforcer. Our current approach does not support the checking of protocols or specific method calls. Thus, the variants one, two, six, and seven are not considered for the test. Moreover, the object interaction of these variants is similar to some variants of
### Security Pattern Detection Tool

<table>
<thead>
<tr>
<th>Authentication Enforcer</th>
<th>Secure Pipe</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>S8</td>
</tr>
<tr>
<td>A2</td>
<td>S6</td>
</tr>
<tr>
<td>A6</td>
<td>S11</td>
</tr>
<tr>
<td>A7</td>
<td>S10</td>
</tr>
</tbody>
</table>

Table 9.1 – Variants using the same object configuration.

<table>
<thead>
<tr>
<th>Security Pattern</th>
<th>Variants</th>
</tr>
</thead>
<tbody>
<tr>
<td>Authentication Enforcer [464]</td>
<td>4</td>
</tr>
<tr>
<td>Information Obscurity [437]</td>
<td>5</td>
</tr>
<tr>
<td>Secure Pipe [464]</td>
<td>11</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>20</strong></td>
</tr>
</tbody>
</table>

Table 9.2 – Number of security pattern variants used for tool testing.

The Secure Pipe pattern (Table 9.1). For this reason, if any of the secure pipe pattern variants are found, the analyst can check the pattern instance manually for authentication settings. Analogously, the variants two, three, four, and five of the Information Obscurity pattern are represented by the object configuration of variant two for our test. Finally, 20 variants are selected to test the COPG approach (Table 9.2).

#### Implementation

To deal with the aforementioned distinct variants for each security pattern, we implemented checkers for each variant. They contain required code objects and the connections, using the input enhancement of COPGs described in Section 9.3.2. With the help of this input definition, our analysis detects interacting objects describing security patterns. For newly supported variants or patterns, new checkers can be written and plugged into our pattern identification (see Section 9.3.2).

#### 9.5 Summary

We have seen in Chapter 8 that security patterns often consist of several objects to provide a specific security feature. In this chapter, we have depicted the challenges of analysts in understanding multiply-interacting objects in the source code. The OPG concept, which captures possible sequences of operations on one particular object, has been enhanced to deal with several interacting objects. We integrated the thereby newly formed COPG approach into an analysis tool to detect security patterns consisting of several objects. This tool is based on the Soot analysis tool-suite and considers the specific Android framework concepts to improve the analysis result. In the next chapter our approach is evaluated with 25 real-world Android apps.
TOOL EVALUATION

The previous chapter has described our detection tool for interacting objects based on security pattern participants. This chapter deals with the feasibility evaluation of our approach. The evaluation is tripartite. The case study evaluating our detection approach is described in Section 10.1. Section 10.2 describes the usage of our tool by inspecting one specific app of our benchmark to check on a priorly detected implementation flaw. Section 10.3 presents a discussion including the obtained feedback from six software security experts from the SAFECODE organization, a non-profit organization, which serves as a forum for software assurance in code. We conclude in Section 10.4.

10.1 Case Study

The case study is based on the guideline described by Kitchenham et al. and includes the following tasks: define the hypothesis, define the method of comparison, select projects, analyze and report the results [286]. In the following, we use these tasks to describe the evaluation of our approach described in Chapter 9 with 25 security-critical Android applications from Google Play.

10.1.1 Defining and Validating the Hypothesis

According to Krämer and Prechelt, two questions arise when a design recovery system that detects patterns is given [293]:

1. Which fraction of all pattern instances is found?

2. Which fraction of the output consists of false positives?

Both questions can be answered with quality measures from information retrieval called precision and recall [235]. Precision and recall can be calculated as follows: $\text{precision} = \frac{T}{F}$ and $\text{recall} = \frac{T}{A}$, where $F$ is the number of all pattern instances found by our tool and $T$ is the number of the truly detected pattern instances. $A$ is the number of pattern instances implemented in the application.
10.1.2 Method of Comparison

The baseline against which we compared our tool was to carry out the first three steps discussed in Section 9.2 manually with some limited tool-support from an Integrated Development Environment (IDE), such as Eclipse. As a preparation of this evaluation, we first employed the Jadx tool [451] to retrieve human-readable source code from the APK files. Thereafter, we searched for classes and API calls that constitute a security pattern as starting points using search features provided by the IDE (see also step 1 in Section 9.2). Then we backtracked the configuration of security-relevant objects following the call hierarchy (see also step 2 in Section 9.2). On doing this step, we also traced back parameters of API calls to understand interactions of the different Java objects that are involved in the implementation of the pattern (see also step 3 in Section 9.2). Our comparison was based on the evaluation of our tool’s precision. Hence, we did not perform a manual security analysis of the detected instances (step 4 in Section 9.2).

Comparison  Following the procedure above, the selected apps (see below) were manually skimmed to collect all possible pattern instances $A$. After that, the COPGs are inspected to retrieve the number of true patterns $T$ and to identify possible false positive pattern instances.

Challenges and Limitations  Due to the fact that libraries are directly integrated into an APK on build-time [189] and often contain unused code, we inspected all packages within an APK. As not all features of an integrated library may be used, we checked whether a manually-detected pattern was called. Moreover, only pattern instances that match the selected variants in Section 9.4.3 were counted. Since our manual detection process was focused on the existence of security patterns, we did not assess their correct usage or search for other possible security problems of the apps, although this is certainly possible and desirable with our approach. In Section 10.2, we demonstrate with the help of an example app how to employ COPGs for such an assessment.

Holistic Information  When no pattern instances were found with the procedure mentioned above, we tried to identify missing security pattern instances based on the viewpoint of security requirements. Therefore, the decompiled code was additionally searched for common security-relevant keywords, e.g. “login”, “auth”, and “encrypt”, and used the search results as starting points for a brief manual detection. This information provides a holistic overview of implemented security features within the analyzed app with regard to the security requirements described by the selected patterns.

Besides evaluating the precision of the proposed tool the execution time for the automatic pattern detection was also measured. The automatic detection ran on a MacBook Pro with Intel Core i5 (2.8 GHz) and 16 GB RAM.
10.1.3 Selecting Apps

The discovery of appropriate Android applications that are suitable for the case study was challenging because they had to fulfill different criteria. First, the apps had to be security-critical because we expected that they contained a relevant set of security-pattern instances within their implementation. Therefore, we inspected the app’s permissions to make sure that it is using the "full Internet access" permission. Furthermore, we surveyed the description to find out whether the app establishes a connection to another service over the Internet. Second, the Java code of the apps should not be (too heavily) obfuscated; otherwise, the effort of manual inspection would significantly increase, which might even falsify our study results. Third, the apps had to be analyzable by the Soot tool. On account of this, the apps were not allowed to be built with specialized frameworks such as Cordova [480] and dependency injection APIs, e.g., Google Guice [181]. Supporting these programming concepts would incur substantial engineering effort. For example, analyzing code that uses dependency injection requires dynamic analyses, whereas the focus of our approach lies in static analyses. The analysis of Cordova/JavaScript apps would require us to implement JavaScript analyses, which is a different topic.

Selection

42 apps were selected from Google Play. Unfortunately, due to the aforementioned problematic factors, many interesting apps had to be excluded from the base set. In total, ten of them use dependency injection frameworks, three use Cordova/JavaScript, two apps are heavily obfuscated, and two other apps could not be processed due to internal Soot failures (Table 10.1).

Benchmark

The remaining 25 apps form a benchmark, which includes non-trivial applications consisting of some hundred lines of code (Table 10.2). These Android apps come from different application areas, such as cyber-physical systems, business, e-commerce, document management, job center, health tracking, mobile payment, and telecommunications. Due to the security-critical nature of the selected apps—some apps are privacy-critical, such as Fitbit and Nike+ FuelBand, as they process personal health data—and the fact that most of them have been published by large vendors, we expected that these apps contained different (interesting) security features. In contrast to other existing benchmarks, such as DroidBench [53] or Stanford SecuriBench Micro (for JEE) [309] our benchmark consists of real-world applications instead of targeting Android-specific challenges like correctly modeling an application’s lifecycle. We are aware of the fact that some Android applications come from the same vendor, e.g., SAP. On inspecting the code, however, one can see that different developer teams have developed them because different libraries and programming styles are used.

Subsequently, the selected apps are introduced briefly to give the reader a better overview of the test set. The SAP Business One app provides mobile access
Problematic Factor | Android App
--- | ---
Uses dependency injection | Bosch Smart Home [394]
 | Magenta CLOUD [479]
 | Commerzbank Banking [91]
 | COQON Smart Home [93]
 | Android Pay [191]
 | Skype - free IM & video calls [452]
 | Amazon Shopping [18]
 | Microsoft OneDrive [335]
 | Microsoft Outlook [336]
 | innogy SmartHome [261]
Uses specialized frameworks | Magenta SmartHome [176]
 | Bosch Remote Security Control [57]
 | Site Monitor [58]
Contains heavily obfuscated code | Home Connect [243]
 | Google [192]
Produces internal Soot failure | Microsoft Authenticator [334]
 | Samsung My Knox [417]

Table 10.1 – Excluded Android applications due to identified problematic factors.

to SAP’s integrated business management application for small-sized enterprises. Managers, executives, as well as marketing personnel can access reports, administer contacts, and perform marketing and service activities. The second SAP app called **SAP BusinessObjects Mobile** is one of SAP’s core apps. It allows remote access to business intelligence data of an enterprise to make informed business decisions. Another selected SAP app (**SAP System Monitoring**) allows a technician to monitor availability, performance, and exceptions of important technical systems. The fourth SAP app **SAP Mobile Documents** allows business customers to store and view business-critical documents on Android devices securely; the documents are fetched from a remote enterprise repository.

The **IBM Mobile Client** enables one to securely connect Android devices to an organization’s network to receive services such as email and recommended apps. Additionally, the app enables the configuration of security settings to protect organizational data on the device.

With the **MagentaSERVICE** app, Telekom customers can administrate their landline accounts, e.g., viewing invoice amounts, establishing call redirection, and viewing contract information. **Mein Blau** and **Mein O2** are two different telecommunications apps from Telefónica. Furthermore, the German Telekom’s **Online Manager** app for Android was selected, which provides access to Telekom WLAN hotspots.

Honeywell’s **MB - Remote Control V2** app allows a customer to control her alarm system remotely. With the **Bosch Control** app, an Internet-connected programmable smart thermostat for heating and hot water system control can be operated, whereas Bosch’s **Remote Security Control Plus** app allows users
to control their security systems remotely from their devices. The Samsung SmartHome, Start SmartHome, and the Nest apps give mobile access to the smart home systems from Samsung, RWE, and Nest Labs (a subsidiary of Google), respectively. The Siemens SIMATIC WinCC Sm@rtClient app allows one to remotely access industrial controllers and the Viewer For Panasonic IP Camera app allows a customer to view and control Panasonic IP Cameras remotely.

The Android version of the Adobe Acrobat Reader and Adobe Sign were included in the Benchmark; both apps are security-critical as they, for example, connect to the Adobe Document Cloud and may process sensitive documents.

With the DB Navigator app one can buy mobile phone tickets, real-time information, delay notifications and personal travel details for trains of the Deutsche Bahn. The benchmark contains additionally the Jobbörse app published by the German job office, which can be considered privacy-critical as it provides information about unemployed persons.

Table 10.2 – Selected Android apps for the case study with their version number and category in Google’s Play market (B=Business, C=Communication, H=Health & Fitness, L=Lifestyle, M=Maps & Navigation, P=Productivity, S=Social, T=Tools).

<table>
<thead>
<tr>
<th>Android App</th>
<th>Version</th>
<th>SLOC</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>MB - Remote Control V2 [244]</td>
<td>02.11</td>
<td>20,133</td>
<td>L</td>
</tr>
<tr>
<td>SIMATIC WinCC Sm@rtClient [448]</td>
<td>01.00.01.00</td>
<td>37,665</td>
<td>P</td>
</tr>
<tr>
<td>Viewer For Panasonic IP Camera [365]</td>
<td>3.3</td>
<td>43,906</td>
<td>P</td>
</tr>
<tr>
<td>SAP System Monitoring [423]</td>
<td>1.0.17</td>
<td>50,038</td>
<td>B</td>
</tr>
<tr>
<td>Start SmartHome [410]</td>
<td>1.0.0</td>
<td>64,514</td>
<td>P</td>
</tr>
<tr>
<td>SAP Business One [421]</td>
<td>1.2.3</td>
<td>80,180</td>
<td>B</td>
</tr>
<tr>
<td>IBM Mobile Client [255]</td>
<td>9.0.40093.1</td>
<td>101,998</td>
<td>B</td>
</tr>
<tr>
<td>Jobbörse [65]</td>
<td>1.4.10-FINAL</td>
<td>116,539</td>
<td>B</td>
</tr>
<tr>
<td>Nike+ FuelBand [351]</td>
<td>1.3.1</td>
<td>125,442</td>
<td>H</td>
</tr>
<tr>
<td>SAP Mobile Documents [422]</td>
<td>1.5.2</td>
<td>133,836</td>
<td>B</td>
</tr>
<tr>
<td>SAP BusinessObjects Mobile [420]</td>
<td>6.2.11</td>
<td>286,382</td>
<td>B</td>
</tr>
<tr>
<td>Bosch Control [56]</td>
<td>3.4.4</td>
<td>277,773</td>
<td>T</td>
</tr>
<tr>
<td>DB Navigator [109]</td>
<td>15.10p04.01</td>
<td>311,575</td>
<td>M</td>
</tr>
<tr>
<td>MagentaSERVICE [178]</td>
<td>6.0.2</td>
<td>387,439</td>
<td>C</td>
</tr>
<tr>
<td>Online Manager [177]</td>
<td>5.0.3.82</td>
<td>389,391</td>
<td>T</td>
</tr>
<tr>
<td>Mein Blau [477]</td>
<td>1.0.3</td>
<td>391,273</td>
<td>P</td>
</tr>
<tr>
<td>Fitbit [166]</td>
<td>2.34</td>
<td>395,400</td>
<td>H</td>
</tr>
<tr>
<td>Mein o2 [478]</td>
<td>6.2.0</td>
<td>404,606</td>
<td>P</td>
</tr>
<tr>
<td>Nest [259]</td>
<td>5.8.1.1</td>
<td>407,749</td>
<td>L</td>
</tr>
<tr>
<td>Adobe Sign [3]</td>
<td>2.7.0</td>
<td>455,646</td>
<td>B</td>
</tr>
<tr>
<td>Ebay - Buy/Sell &amp; Save Money [119]</td>
<td>5.5.1.17</td>
<td>458,867</td>
<td>S</td>
</tr>
<tr>
<td>Instagram [262]</td>
<td>10.11.0</td>
<td>538,829</td>
<td>S</td>
</tr>
<tr>
<td>Samsung Smart Home [418]</td>
<td>1.1072.27.110</td>
<td>568,855</td>
<td>T</td>
</tr>
</tbody>
</table>

139
We also included the eBay app for buying and selling goods on the eBay the marketplace via smartphone or tablet computer and the Instagram app for Android in our test set. Finally, our set contains the fitness tracking apps Nike+ FuelBand and Fitbit for fitness trackers developed by Nike and Fitbit, respectively.

10.1.4 Analyzing the Results

Table 10.3 shows the result of the case study. For each app, information on the automatic detection and self-implemented instances are given. Besides the measured detection time of our tool, the automatic detection information embraces for each security pattern the manually detected instances ("all"), the instances found by our tool ("found"), and the correctly detected instances ("true"). Based on these three columns the precision and recall per app are calculated according to the definition in Section 10.1.1. In total, 248 security pattern instances were manually detected in the selected Android applications. The manual analysis took about four hours per app, leading to a net analysis time of about 100 hours.

We could not find any security patterns within the Start SmartHome app. After a closer look, we saw that the app simply provides a pre-configured WebView object that displays the website of the target smart home system. For six apps (MB - Remote Control 2, Viewer for Panasonic IP Camera, IBM Mobile Client, Bosch Control, Nest, and eBay - Buy/Sell & Save Money) all manually identified Information Obscurity and Secure Pipe pattern instances were detected correctly by our tool. Unfortunately, our tool missed several pattern instances which we detected manually. Therefore, we took a closer look at some of the missed instances and inspected them again to identify possible problems with our approach.

SIMATIC WinCC Sm@rtClient: Our tool correctly detected three Information Obscurity patterns, but we detected one more Secure Pipe pattern during the manual review in the SIMATIC WinCC Sm@rtClient. The undetected Secure Pipe pattern results from missing callgraph information for the SSLContext object that returns the SSLSocketFactory, which in turn is used to create a Socket object.

SAP BusinessObjects Mobile: Our tool recognized 16 Information Obscurity patterns in this app. Two Information Obscurity patterns are used within a Runnable object’s run method which should be called from the object’s start() method. This call was missing in the constructed callgraph. Besides, our tool detected 14 of 17 Secure Pipe pattern instances in the app. The three missing instances seem to be runtime- or configuration-dependent. Unfortunately, the class that calls the pattern instances can only be partly decompiled. Thus, we were unable to check the problem in depth.

SAP Business One: This app contains four Secure Pipe patterns, but our tool only detected three of them. The missed pattern is used within a Runnable object’s run() method, which should be called from the object’s start method. Again, this call was missing in the constructed callgraph.
<table>
<thead>
<tr>
<th>App</th>
<th>Auth. Enforcer</th>
<th>Inform. Obscurity</th>
<th>Secure Pipe</th>
<th>Self-Implemented Instances</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>found</td>
<td>true</td>
<td>found</td>
<td>true</td>
</tr>
<tr>
<td>MB - Remote Control V2</td>
<td>0 0 0</td>
<td>2 2 2</td>
<td>2 2 2</td>
<td>100</td>
</tr>
<tr>
<td>SIMATIC WinCC Sm@rtClient</td>
<td>0 0 0</td>
<td>3 3 3</td>
<td>1 0 0</td>
<td>100</td>
</tr>
<tr>
<td>Viewer For Panasonic IP Camera</td>
<td>6 6 6</td>
<td>0 0 0</td>
<td>7 7 7</td>
<td>100</td>
</tr>
<tr>
<td>SAP System Monitoring</td>
<td>0 0 0</td>
<td>6 4 4</td>
<td>1 1 1</td>
<td>100</td>
</tr>
<tr>
<td>Start SmartHome</td>
<td>0 0 0</td>
<td>0 0 0</td>
<td>0 0 0</td>
<td>–</td>
</tr>
<tr>
<td>SAP Business One</td>
<td>0 0 0</td>
<td>0 0 0</td>
<td>4 3 3</td>
<td>100</td>
</tr>
<tr>
<td>IBM Mobile Client</td>
<td>0 0 0</td>
<td>0 0 0</td>
<td>1 1 1</td>
<td>100</td>
</tr>
<tr>
<td>Jobbörse</td>
<td>0 0 0</td>
<td>2 0 0</td>
<td>2 2 2</td>
<td>100</td>
</tr>
<tr>
<td>Nike+ FuelBand</td>
<td>1 0 1</td>
<td>1 1 1</td>
<td>8 5 5</td>
<td>100</td>
</tr>
<tr>
<td>SAP Mobile Documents</td>
<td>1 1 1</td>
<td>2 2 2</td>
<td>3 1 1</td>
<td>100</td>
</tr>
<tr>
<td>Remote Security Plus</td>
<td>2 2 2</td>
<td>10 5 5</td>
<td>9 4 4</td>
<td>100</td>
</tr>
<tr>
<td>SAP BusinessObjects Mobile</td>
<td>0 0 0</td>
<td>18 16 16</td>
<td>17 14 14</td>
<td>100</td>
</tr>
<tr>
<td>Bosch Control</td>
<td>0 0 0</td>
<td>2 2 2</td>
<td>3 3 3</td>
<td>100</td>
</tr>
<tr>
<td>Adobe Acrobat Reader</td>
<td>2 2 2</td>
<td>13 13 13</td>
<td>14 7 7</td>
<td>100</td>
</tr>
<tr>
<td>DB Navigator</td>
<td>0 0 0</td>
<td>5 5 5</td>
<td>9 6 6</td>
<td>100</td>
</tr>
<tr>
<td>MagentaSERVICE</td>
<td>1 0 0</td>
<td>2 0 0</td>
<td>12 12 10</td>
<td>83</td>
</tr>
<tr>
<td>Online Manager</td>
<td>0 0 0</td>
<td>3 3 3</td>
<td>14 12 12</td>
<td>100</td>
</tr>
<tr>
<td>Mein Blau</td>
<td>0 0 0</td>
<td>6 3 3</td>
<td>2 2 2</td>
<td>100</td>
</tr>
<tr>
<td>Fitbit</td>
<td>2 1 1</td>
<td>0 0 0</td>
<td>4 2 2</td>
<td>100</td>
</tr>
<tr>
<td>Mein o2</td>
<td>0 0 0</td>
<td>6 3 3</td>
<td>2 2 2</td>
<td>100</td>
</tr>
<tr>
<td>Nest</td>
<td>0 0 0</td>
<td>1 1 1</td>
<td>10 10 10</td>
<td>100</td>
</tr>
<tr>
<td>Adobe Sign</td>
<td>0 0 0</td>
<td>4 4 4</td>
<td>12 10 10</td>
<td>100</td>
</tr>
<tr>
<td>Ebay - Buy/Sell &amp; Save Money</td>
<td>0 0 0</td>
<td>3 3 3</td>
<td>4 4 4</td>
<td>100</td>
</tr>
<tr>
<td>Instagram</td>
<td>0 0 0</td>
<td>0 0 0</td>
<td>9 7 7</td>
<td>100</td>
</tr>
<tr>
<td>Samsung Smart Home</td>
<td>0 0 0</td>
<td>8 5 5</td>
<td>6 5 5</td>
<td>100</td>
</tr>
</tbody>
</table>

Table 10.3 – Our case study results per analyzed app with the number of instances found by manual review and our tool as well as the number of true instances.
**MagentaSERVICE:** Our analysis detected 12 of 12 *Secure Pipe* pattern instances, but two detected instances were detected twice. The missing two instances were used within annotated code, which generates plumbing code at runtime time [265]. The missed *Information Obscurity* patterns were used by a self-written *ExecutorService*, which could not be found in the call graph generated by Soot. Moreover, our tool missed an *Authentication Enforcer* pattern, which we detected manually. This pattern uses the Android *AccountManager* and *Account* object to create and manage accounts. Unfortunately, the call graph seemed to be incomplete for these objects. Thus, we could not detect the interaction between the *AccountManager* and the *Account* object.

**Mein Blau und Mein o2:** The two Telefónica apps share some libraries according to the package names within the APK file. Thus, the analyses achieve similar results. The tool detected the manually identified *Secure Pipe* instances and three of the six *Information Obscurity* instances. Due to the selected context-insensitive pointer-analysis (Spark) the pointer information for context-dependent newly-created objects returned by a method are treated as one single object. Consequently, this object is always returned by the method instead of a new object for each method return (for further details see [459]).

**Remote Security Plus:** Ten *Information Obscurity* instances were detected manually; the tool recognized five of them. Three instances of them were missed due to the insensitive pointer-analysis previously mentioned for the “Mein Blau” and “Mein o2” apps. The other two *Information Obscurity* instances are used within a *Runnable* class, in which Soot misses some information as described earlier. This “Runnable” problem also applies to the five missed *Secure Pipe* instances.

**Samsung Smart Home:** The pattern detection found five *Information Obscurity* of eight expected instances within the app. The three missed instances stem from the selected points-to analysis in Soot, which is not context-sensitive. This also applies to the missing *Secure Pipe* instance. Thus, the instances cannot be found due to a lack of pointer information. Moreover, we detected six self-implemented *Secure Pipe* instances. They are based on the Apache HttpComponent objects which are integrated into the Android Framework and additionally added objects located in the package `org.shaded.apache.http.client.methods`.

We can summarize the following points that may lead to the missing pattern instances:

- Self-written implementations (*ExecutorServices* and runtime- or configuration-dependent implementations) influence the callgraph and points-to analysis. This point cannot be addressed in general due to its individual nature per app.
- Predefined runtime-dependent system services, which generate runtime-dependent information, such as the **AccountManager** which generates dynamically the *Account* object. We do not simulate predefined Android system services, and the static analyses limit our approach.

- The points-to analysis seems to be a good point for improving the results. A context-sensitive pointer analysis such as geomPTA may improve the results [458]. We tested two apps with this pointer analysis, but both of them got an out-of-memory error and the analysis stopped. Using other pointer analyses could be investigated in future work.

In total, the introduced tool identified 201 pattern instances whereof only two instances were false positives. Over all analyzed apps, the tool achieves high precision (99%) and acceptable recall (80%) marks. The recall can be increased by addressing problems that seem to stem from the *Soot* tool. Please see below for a further discussion of false positives and negatives.

Unexpectedly, many manually-written **Authentication Enforcer** pattern instances were found. Since our tool provides a manual configuration option for COPGs (see Section 9.3.2), our tool can be tailored towards supporting application-specific interacting objects. For example, one can define a COPG input specification for the OAuth2-authentication mechanism that has been manually detected within the SAP Mobile Documents app. This authentication-enforcer variant consists of an **OAuthManager** object, which organizes the OAuth2 functionality within the app by using an **OAuthAuthorizationProfile** and an **OAuthAuthenticationProvider** object (Figure 10.1). The OAuth endpoint information and client secret are given in the **OAuthAuthorizationProfile** object and the **OAuthAuthenticationProvider** object creates a **Token** object with the received information of the online service (OAuth access and refresh tokens). To better deal with these customized and scattered security features, also a technique for the detection of pattern fragments is desirable.

![Figure 10.1 – A manually detected Authentication Enforcer variant in the SAP Mobile Documents app.](image)

Nevertheless, on finding interacting security objects, the presented tool decreases the detection time notably compared to the manual detection. Moreover, with its
low false-positive rate the results are promising to support security reviewers at their work.

10.2 Exemplary Usage of COPGs

COPGs shall support a security auditor in the process of security program comprehension. In the following, it is explained how an analyst can employ our prototype within the Eclipse IDE.

We discuss the comprehension task with the help of the COPG shown in Figure 10.2. It depicts an Information Obscurity pattern detected in the SIMATIC WinCC Sm@rtClient app. The COPG consists of two objects; the blue backgrounded icons are statements belonging to a Key object and the yellow ones belong to a Cipher object. Each node provides a tooltip that shows the statement connected to the node. It allows an analyst to browse the graph to a certain statement of interest based on abstract node and statement information. Moreover, the icons are directly linked to the source code. This feature allows an analyst to double-click on an icon such that an editor with the corresponding statement in the source code opens. An auditor can easily find the create node of each object and the connection node, which connects the two objects in the control flows (marked with dashed circles in Figure 10.2b).

An auditor can follow one object, e.g., the Cipher object (step 1), and then use a connection (or shared) node to trace a different object, e.g., the key. In the SIMATIC app (Figure 10.2b), the connection node is the statement cipher.init(Cipher.ENCRYPT_MODE, skeySpec, ivSpec); (step 2) which uses the sKeySpec object, as can be seen by jumping into the code (step 3). Now the auditor can trace the sKeySpec object in the source code or search for the second create node within the COPG (step 4).

The value of the tool lies in decoupling the detection from the actual analysis process such that an auditor can focus on analyses rather than repeatedly browsing through the code. As the COPG contains information at the statement-level, an analyst can track the complete configuration of those objects implementing a security feature. For example, the SIMATIC WinCC Sm@rtClient app contained an implementation flaw in an earlier version because it used the WiFi address to generate a symmetric key. This key, in turn, was used for storing the password of the industrial controller on the phone. After reporting this flaw to Siemens, our tool was used to confirm that the problem has been fixed appropriately. This case shows how our tool can be applied within review tasks of security features, which are implemented in applications. It is important to note that COPGs can be used by an analyst to control the process of code audits by digging into code details to review an object’s specific configuration and then jumping back to the COPG to trace another object.
10.3 – Exemplary Usage of COPGs

(a) A view showing all nodes of the extracted COPG.

(b) A view of the extracted COPG where the API node groups all nodes that belong to the Android API (for clarity reasons).

Figure 10.2 – Screenshots of a detected Information Obscurity COPG in the SIMATIC WinCC Sm@rtClient app. Blue backgrounded icons refer to a Key object and yellow ones to a Cipher object. Dashed circles mark creation and connection nodes in both figures. A double-click on a node allows the user to open the corresponding statement in the source code.
Discussion

In this section, we take a closer look at usage scenarios of COPGs, and the relevance of false negatives and positives rates as additional information for the review process are discussed. Finally, different use cases of the pattern detection with regard to the implementation as well as the architectural level are presented.

All the topics in this section have been discussed with six software security experts from SAFECode in a video conference session. The SAFECode members are software security experts working for global enterprises, such as Siemens, Google, Adobe, Microsoft, Veracode, and Symantec [455]. First, our approach was presented, and the exemplary usage of our tool with the help of our Eclipse plug-in and the Siemens SIMATIC WinCC Sm@rtClient app was demonstrated. The demonstration was carried out in an interactive fashion such that we obtained immediate feedback. The whole demonstration and discussion took about one hour. Finally, we discussed more general questions concerning our approach. The discussed questions are given in Appendix F.

Usage Scenarios

The SAFECode members gave the following scenarios where our pattern detection can be deployed. First, one can use the detected pattern instances (COPGs) for low-level security analyses, e.g., checking whether the parameters for encryption (keys, IVs) are of adequate strength or hostname verification of TLS clients is done correctly. This step can even be conducted without considering the high-level security patterns as described in Chapter 8. Second, if architectural diagrams are available (e.g., data flow diagrams as used by Microsoft’s Threat Modeling), a security analyst can manually note information about implemented security patterns directly in architectural descriptions as shown in Figure 10.3. This step gives an auditor more
information on the security posture of the entire application under analysis. Finally, information about implemented security patterns can also be used for requirement tracing in the final code audit [68]. Conceivable is also to cluster the detected security patterns of several analyzed systems within a company's application landscape to obtain an overview of security precautions exposed by the running applications.

Section 9.3.2 has already described that our tool can be configured with XML-based descriptions. These descriptions can be shared among users of a software vendor to provide a common standard for starting and assessing software security within the enterprise. Also, it is conceivable to provide an online collection, which contains security-relevant COPG descriptions and which is freely available to users interested in assessing applications with regard to security.

10.3.2 False Positives and Negatives

Our tool produces both false positives and negatives, however, with a considerably better false-positive rate (99% precision compared to 80% recall). False positives give the analyst a false sense that certain security features are used, which is not the case. At least, false positives may puzzle the analyst. As the false-positive rate, however, is comparatively low, this point does not pose a threat to our approach.

False negatives are pattern instances that our tool misses. Nevertheless, the tool detects a significant portion of correct pattern instances in code (199 of 248), which saves time and effort compared to solely carrying out a manual analysis. Without the tool, an analyst must manually note down trace hierarchies. Otherwise, she would end up visiting the same code locations several times during the review, e.g., if the security-relevant code is scattered across several classes/methods (see also Section 9.2).

According to the security experts from SAFECode, the fact that expected pattern instances are missing is an important information for a security analyst. If she finds fewer pattern instances than expected, she may focus the security analysis on those particular security requirements. If it turns out that self-implemented security features have been implemented (in general a bad practice), a more thorough manual assessment is required. For example, the many self-implemented Authentication Enforcer patterns, which do not use well-tested authentication features (such as the Android account manager API), must be analyzed in more detail.

10.4 Summary

The pattern-variant detection has been evaluated in a case study with 25 non-trivial Android applications and shows acceptable results on precision and recall. Besides, this benchmark of benign Android apps can be used by researchers in other projects.

Our tool can be used for reviewing Android applications at earlier development stages and before a software will be released. It also provides support for security audits by automatically detecting source code locations of implemented security features and tracing a security-relevant object’s configuration. Analysts can
utilize the visual representation of connected objects to understand security-object interactions. We see our tool as an enhancement for reviewing Android applications rather than a replacement for common vulnerability detection tools, which only support highlighting of erroneous source code locations. In the end, this work can be regarded as a first step towards supporting security analysts with regard to program comprehension.
Part IV

Finale
The related research of this thesis can be divided into several areas. First, works on security patterns with the subtopics classification, collection, and description quality are presented. Second, design and security pattern detection in software systems followed by the detection of security concepts on the architectural level are described. Then, analysis techniques related to our COPG approach are depicted. Finally, as this work also focuses on analyzing Android applications (Chapter 8 and 9), comparable static analysis approaches tailored towards Android specialties, such as the app lifecycle simulation, are presented briefly in Section 11.5.

11.1 Security Patterns

This thesis tackles the security pattern topic in several ways and related work on the topics has already been presented in several sections before. Thus, the next sections present briefly the topics collection, classification, and description quality.

11.1.1 Collection

Security patterns are often scattered across several scientific works. Thus, for example, the SecurityPatterns.org website collects several security pattern publications to provide an overview of the available security patterns [439]. Moreover, three main works in book-form have been released [144, 437, 464] and some other pattern collections have been published (Table 11.1). Heyman et al. [238] collected the largest number of security patterns (220). Unfortunately, they did not publish their catalog.

All catalogs only cover parts of all published security patterns since 1996. This thesis collects 528 published patterns from the years 1996 to 2016 and provides a holistic overview of these patterns.
11.1.2 Classification

Security patterns can be used like design patterns to structure software. However, there are security patterns which describe security requirements or processes for enterprises, e.g., the Enterprise Architecture Management Pattern [136]. Thus, many different classifications exist. They have already been presented and discussed extensively in Section 4.2. Concluding, the discussed classifications only consider a few patterns for their classification.

In this thesis, two classification approaches for all collected security patterns with regard to “application” and “software reengineering” concerns have been presented (Section 4.4 and Section 4.5). The first classification scheme is based on all collected security patterns and shaped towards the selection by application domains, which is relevant for researchers and practitioners who are interested in security patterns. The second classification shows in detail which security and implementation forces the software-security patterns have.

11.1.3 Description Quality

With the increasing number of publications, the quality partially decreases [238]. Some works depict that the quality of security pattern descriptions differs and is sometimes even not sufficient. Yoshioka et al. examine patterns according to their ease of use, effectiveness, and sufficiency in general and give some examples [539].

Halkidis et al. [213] inspect the patterns collected by Blakley et al. [50] according to the ten principles of building secure software [506], three software development problems with regard to security (buffer overflow, poor access control, and race condition), and the STRIDE model.

A more general approach that is not related to security has been presented by Laverdiere et al. [300]. They utilize the House of Quality evaluation framework to evaluate twelve security patterns. Moreover, they show that the inspected patterns have many undesired properties such as lacking generality, or under- or over-specification.
The appropriateness and quality of security-pattern documentation has been examined by Heyman et al. [238]. They use a scoring system to measure the quality of a description element and show that core patterns have medium to high quality. In addition, the guidelines, best practices, and process activity patterns have a documentation quality between low and medium.

Security pattern investigations with regard to quantity and in particular their quality have been conducted for only a few security patterns. In this thesis, the description quality has been assessed by inspecting the usage of UML figures and given code examples of all collected software-security patterns.

11.2 Pattern Detection

Design patterns are often used in software development to design software systems. Reengineering these patterns gives maintenance programmers and analysts specific information on implemented solutions. This section focuses on the two pattern types discussed in this thesis: design and security patterns.

11.2.1 Design Patterns

In the area of design-pattern detection several approaches with different analysis styles exist. The used detection techniques span many areas, such as fuzzy sets, machine learning, constraint satisfaction, and graph theory. A comprehensive overview of the available approaches give Dong et al. [112, 113], Rasool and Streitferdt [379], Ampatzoglou et al. [20], and Al-Obeidallah et al. [8]. Thus, only some approaches are listed in the following to give a brief overview:

**Structural**: Pattern instances are detected by static program information. Such analyses inspect inter-class relationships and method invocations. [28, 30, 33, 47, 100, 141, 198–200, 278, 293, 315, 362, 453, 497, 509, 511, 548].

**Behavioral and structural**: These analyses encompass behavioral program aspects additionally to the structural analysis information. They are extracted using static and dynamic analysis techniques. [12, 24, 104, 237, 275, 301, 314, 350, 440, 466, 497, 498, 515, 524, 542, 543].

**Structural and semantic**: Enhancing structural analysis with semantic information aims to decrease the false positive rate of the structural detection. Semantic information is, for example, considering naming conventions, annotations or metrics to retrieve role information for pattern components. [24, 380, 440, 489].

**Structural, behavioral and semantic**: Analyses that use all three types aim to combine the benefits of each analysis type to reduce the number of incorrectly detected patterns instances. [51, 111, 114, 445, 446].

Some of these approaches use analyses on graphs to detect patterns. The OPG is also a kind of graph, which we use to detect patterns. Thus, the next paragraphs
will present approaches using a UML or graph representation of the source code in more detail.

In the approach of Seemann and von Gudenberg, a compiler collects method calls and inheritance hierarchies from Java code [440]. The resulting graph is then filtered to detect Strategy, Bridge, and Composite design patterns.

The Columbus tool analyses C++ source code and uses an internal scheme that captures the C++ language at a low-level representation (Abstract Semantic Graph (ASG)) enhanced with higher-level elements such as semantic of types [141]. The main program information such as classes, attributes, and their relationships are extracted from the ASG, written in PROLOG format, and passed to another tool called Maisa. Maisa does the automatic design pattern detection with its integrated pattern library.

Balanyi and Ferenc introduce an XML–based language (Design Pattern Markup Language (DPML)) to provide the ability of customizable pattern descriptions for the detection [30]. They use the Columbus tool to extract the main program information as described before. Then, the DPML-based pattern description file is loaded into an XML DOM tree. Finally, the detection algorithm matches the DOM tree to the ASG to identify implemented pattern instances.

A comparable approach has been presented by two other works [445, 446]. They use an Abstract Syntax Tree (AST) data structure built with static analysis information. They extract data and control-flow information of the AST to obtain static behavioral information for the pattern detection.

The approach presented by Tsantalis et al. [497] is based on the idea that a class diagram is a directed graph that can be mapped into a matrix. The represented information in the matrix depends on the characteristics of the pattern that should be detected. Moreover, the pattern which should be detected is represented as matrices. In the next step, inheritance hierarchies are built to inheritance trees. Then the inspected software system is split into subsystems consisting of classes belonging to one or more hierarchies. An algorithm is applied to calculate the similarity between the subsystem matrices and the pattern matrices to extract the pattern instances within the subsystems.

Yu et al. present an approach based on sub-patterns [542, 543]. First, they create sub-patterns of GoF patterns. Then, they begin with the transformation of source code into directed graphs. Such a graph consists of nodes representing classes and edges and their weights model their relationships. The pattern instances are identified by a subgraph discovery method where the extracted subgraphs are matched with predefined structural features. Finally, behavioral characteristics of the GoF patterns are considered for obtaining the final pattern instances.

One of the latest approaches in this area focuses on detecting new pattern variants of the GoF patterns with four already available pattern detection tools [513]. As far as we know, there exists no approach or similar technique that uses OPGs for the detection nor is tested with Android apps.
11.2.2 Security Patterns

VanHilst and Fernandez [503] discuss the possibilities to detect security patterns using reverse engineering like the approach by Niere et al. [350]. They identify some problems that may occur during detection, but they do not describe a practical approach.

Most approaches to security-pattern recovery have been carried out manually. Two approaches conduct the design recovery manually without the help of a tool that supports program comprehension through increasing the abstraction level of the software [202, 208]. Hafiz et al. inspect the architecture of Sendmail according to its security pattern usage and detect that the qmail architecture provides a high level of security by using security patterns [208]. This work was followed up by a brief comparison of five architectures of mail transfer agents with regard to security pattern usage [202]. In contrast to that, the work in Chapter 7 has the reverse engineering point of view. It uses an abstract representation based on the software implementation for gaining architectural information encapsulated by security patterns.

The first automated detection approach based on ordered matrix matching has been presented by Alvi and Zulkernine [16]. They use the relations and objects of a security pattern given in class and sequence diagrams to create the ordered matching matrix and enhance an existing subgraph isomorphism algorithm to identify patterns. Their approach depends on given class and sequence diagrams as well as standard implementations of security patterns matching the given diagrams. They test their detection tool with four Java-based applications and three security patterns (Single Access Point, Security Session, and Authenticator). Each application has only been tested whether a specific security pattern is contained, e.g., the Simple Android Instant Messaging app has been only searched for the Single Access Point pattern. In comparison, our tool evaluation embraces 25 different Android applications and measures, besides their existence, the precision and recall for the detected patterns. Furthermore, Alvi and Zulkernine only highlight the existence of a pattern and do not collect further information, such as object traces, which are also useful for reengineering and reviewing tasks. On the contrary, our approach introduces an automatic detection method for security patterns based on object interaction which is configurable and integrated in an IDE. Moreover, we depict in this thesis how security patterns of interest can be identified without having class and sequence diagrams.

11.3 Detecting Security Concepts at Architectural Level

Only a few works deal with reverse engineering the security architecture out of code and are focused on security. An approach to the automated verification of UMLsec models has been presented by Jiürjens and Shabalov [273]. UMLsec extends the Unified Modeling Language (UML) and integrates security-related information
into UML specifications [528]. They provide the ability to verify automatically UML models with regard to modeled security requirements. The precondition to this is that there is a UML-modeled architecture for the software system.

Ryoo et al. present a basic approach to detecting architectural constructs and properties that make software less secure [411]. They describe a three-stepped approach to detect software architectures that are either resilient or vulnerable to security attacks.

Karppinen et al. conduct a case study in which they detect a security back door [277]. For their case study, they completely removed the security check of the program to be inspected. Then they used the Software Architecture Visualization and Evaluation tool to detect the back door in the program. They used static as well as dynamic information and compared the resulting information with the results of the correct implementation.

Another idea of employing the software architecture using static security analysis has been described by Sohr and Berger [456]. They use the RFG to check policies and permissions on the JEE and the Android platform. Furthermore, Berger et al. present an approach to extracting a software’s security architecture using Data Flow Diagrams (DFDs) for Java-based programs [45]. Later, their approach has been extended to Android apps [46].

Similar to the approach mentioned above Jung et al. [271] propose a method to recover the software architecture of Service-Oriented Architecture (SOA) systems. They use static security analysis techniques to extract a software architecture and security-related information from available system artifacts. They tag architectural views with priorly defined security rules of a knowledge base. A comparable approach has been introduced by Guan et al. [195]. They describe an approach to extracting system models from source code and identify security relevant artifacts by matching them to a built-in security artifact base.

In contrast to the presented approaches, the presented manual recovery approach within this thesis uses the hierarchical reflexion method and aims explicitly to detect security patterns (Chapter 7).

11.4 Related Techniques

This section presents techniques that are closely related to the Connected Object Process Graphs (COPGs). They have a similar representation, extraction technique or similar way of application. The presented techniques are often available for static and dynamic program information. As we emphasize on static analysis in this thesis, the following sections focus on approaches based on static analysis.

11.4.1 Static Trace Extraction

The ideas of static trace extraction are first of all investigated by students of the University of Stuttgart [221, 508]. Later on Eisenbarth et al. published the extraction technique and representation of static traces as OPG [124, 126]. Besides the static
trace approaches, dynamic and hybrid (static and dynamic) object trace recovery techniques appeared [376, 431].

The implemented tool based on OPGs was influenced by the works of Eisenbarth and Quante [126, 376]. We extended the representation concepts of the OPG and the DOPG (Dynamic Object Process Graph) with concepts representing the connection of OPGs (Section 9.3). In contrast to the DOPG approach, our work is entirely based on static analysis.

All in all, the described techniques for OPG and DOPG extraction are limited to tracing one single object. OPGs capture the trace of one object that is initialized with the `new` statement in source code only. The instantiation of objects can happen deep inside the API implementation, but analysts are only interested in the point of existence in application code. Thus, new starting points for OPG extraction are added to capture the static and sequential creation of objects to tackle this issue.

### 11.4.2 Slicing

Slicing can be performed by static or dynamic analysis techniques. A comprehensive overview of program slicing techniques can be found by Lucia [313], Sasirekha et al. [426], and Singh and Singh [449]. Therefore, this section focuses on static analysis techniques as we use static program information for the OPG extraction and COPG generation.

Program slicing has been introduced by Weiser [519–521]. This technique uses control and data flow dependencies to identify expressions that influence variables. Slicing can be done in two directions forward and backward:

**Backward slicing** identifies all expressions that influence a given variable in a certain source location.

**Forward slicing** identifies all expressions that are influenced by a given variable.

The original slicing algorithm by Weiser operates on statements within procedures and does not take information about crossing boundaries of procedure calls into account [520]. Recent slicing techniques use a program dependence graph (PDG) in terms of program reachability [163, 295, 361]. In a PDG nodes are corresponding to statements and control structures. It is a directed graph with edges carrying data and control flow information. The PDG-based algorithm has been extended by Horwitz et al. to build a System Dependence Graph (SDG) [245]. In contrast to a PDG the SDG “incorporates collections of procedures (with procedure calls) rather than just monolithic programs” [246] to enable context-sensitive interprocedural-slicing.

Slicing techniques have also been extended to the needs of object-oriented programs by modeling parameter passing and using static pointer analyses [22, 76, 79, 217, 299, 487, 492]. In addition, Liang and Harold introduce a specialized form of object-oriented slicing named *object slicing* focusing on objects “to enabled the user to inspect the statements in the slice object-by-object” [22]. All in all, the presented
object-oriented slicing-variants are still following the original ideas by Weiser, which are based on program slicing for procedural programs.

OPGs can be statically extracted with techniques similar to (forward) program slicing. Then one can select API calls such as doFinal as slicing seeds and transitivity calculate their backward dependences. The problem with such an approach is that it must also follow dependences through the API’s implementation, which is expensive and even inaccurate due to missing information on heap locations [462]. Many dependences are finally lost. For example, slicing the code with regard to doFinal with the help of the WALA slicer [462]

```java
cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
cipher.init(Cipher.ENCRYPT_MODE, key);
return cipher.doFinal(data);
```

yields

```java
cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
return cipher.doFinal(data);
```

The init call, which connects the key with the cipher object, is missing in the slice as no direct data dependence between the init and doFinal calls exists. The actual data dependence is hidden within the implementation of the Java crypto library, i.e., an internal member variable of the Cipher class for the key is used within the doFinal method and WALA misses this dependence. To address this point, one certainly can use the fact that doFinal and init are called on the same object, which can be easily determined by an additional dataflow analysis in this case. However, the init call could have occurred in a different method, which would lead to an inter-procedural pointer analysis, which is known to be expensive [304]. In the end, such an analysis would result in the construction of OPGs for Cipher objects.

The presentation of a slicer’s output is often statement-based. In contrast, OPGs and COPGs only contain the relevant statements to selected object(s), which are browsable through a graph structure. Nodes are typed, which enables an analyst to capture the action related to an object at one glance. In addition, on creating COPGs the algorithm preserves all statically possible variations in code, e.g., a static method that creates a pre-configured Cipher object for all cryptographic actions within an app/software will result in n COPGs. This allows an analyst to inspect all used configurations with the Cipher object.

The difference between the two extraction techniques is given in the notion of relevant expressions. For object tracing an expression is only relevant if it is related to one particular object. In contrast to that, every expression reachable via control and data dependencies is relevant for program slicing.

11.4.3 Type-State Analysis

The type-state analysis is a form of program analysis employed in object-oriented languages programming languages. The published type-state approaches use static [39, 48, 108, 165], dynamic [116], or hybrid (static and dynamic) [52, 54, 346] analyses
to check instances of a given type to valid sequences of operations. In contrast to that, the COPG analysis does not model states or test valid sequences of operations on these states.

Strom and Yemini suggested the first concept of type-state analysis in 1986 [467]. Some years later, a static type-state analysis is integrated into the Fugue tool to analyze Microsoft .NET based programs by DeLine and Fähndrich [108]. The authors assume that a programmer of the tool has annotated the program under test with respect to how method calls of an object change their type-state. Their programming model encompasses aspects, such as down-casting, virtual dispatch, direct calls, subclassing, and aliasing, which are typical for object-oriented programs.

A comparable approach for Java programs has been presented by Bierhoff and Aldrich [48] and Beckman et al. [39]. They describe a modular approach using access permissions with references to allow the type checker to reason about a local reference. Similarly to the work of DeLine and Fähndrich [108], this approach is also based on a preexisting program annotation that contains information about access permissions and type-state changes.

Fink et al. present an automated approach for Java type-state analysis using a flow-insensitive pointer-based analysis followed by flow-sensitive checkers [165]. A static type-state analysis using aliasing and a lifetime dependency analysis for C++ programs has been proposed by Xiao et al. [534].

The approaches described before can only reason about one single object at a time. An approach that can handle multiple interacting objects has been presented by Bodden et al. [54]. They proposed a hybrid type-state analysis that is flow-sensitive on an intra-procedural level and uses a flow-insensitive abstraction of the remainder of the program. Later, they enhance their approach by using an additional backward pass [52]. A comparable inter-procedural approach has been proposed by Naeem and Lhotak [346]. Their approach is also a context-sensitive and flow-sensitive whole-program analysis for multiple interacting objects.

11.4.4 Feature Location

The aim of feature or concept location is to identify an initial location in the source code that corresponds to a specific functionality [49]. Feature location techniques can be distinguished by the type or types of analyses they employ to identify the code that belongs to a feature. The most common types of analyses are based on dynamic, static, textual, and historical information plus their combinations. Dit et al. give an extensive overview of the different approaches [110]. In the following, the common analysis types for detecting a feature location are sketched briefly.

Dynamic feature location relies on collecting information from a system during runtime using dynamic analysis. Many approaches exist using dynamic analyses, some of them are [4, 122, 127, 529, 530, 532]. A comprehensive survey of dynamic feature location approaches is given by Cornelissen et al. [95].

Static analysis techniques use information such as control or data flow dependencies to locate features in the source code. Static feature location techniques
require a dependence graph of the software and software artifacts which serve as a starting point for the analysis. Some works that describe these techniques are given, for example, by Chen and Rajlich [78], Robillard [395, 396], Robillard and Murphy [397], Saul et al. [427], and Trifu [494, 495].

Textual analysis approaches are also used to locate features. These approaches are based on identifiers and comments encoding domain knowledge in the source code. The general idea is that similar words are used for certain features throughout a software system. Thus, implemented features can be identified by finding these words and mapping them to a source code location. The approaches in this area can be divided into three main areas: textual search with grep [364, 531], Information Retrieval (IR) [89, 174, 316, 322, 368, 377] and Natural Language Processing (NLP) [75, 239, 240, 247, 443, 444].

There are also approaches that combine two or more analysis styles to improve the detection correctness. Prevalent is the combination of static and dynamic techniques where the dynamic analysis is used to reduce the search space of program elements that are of interest and then the static analysis ranks the remaining elements or find additional elements [23, 125, 399, 514].

The combination of static and textual techniques is also a good combination to reduce the overestimation of the static analysis or to find additional relevant program elements to the selected elements of the textual analysis [232, 233, 382, 441, 549]. The composition of dynamic and textual techniques can be used to rank program elements by their relevance, or the dynamic analysis can be used to filter code elements for the textual analysis [27, 231, 308, 369]. The only approach combining all of the three analysis types has been presented by Eaddy et al. [118]. All in all, combined analyses produce better results than using a single feature location technique [110].

Besides the aforementioned common analysis types, there are some other approaches that use additionally historical information to locate features. For example, Chen and Rajich [77], Ratanotayanon et al. [381], and Chochlov et al. [84] use textual and historical information of a version-control system. In addition to code versions, three other approaches use artifacts from a project’s archives, such as online documentation and bug reports, to identify features [381, 470, 545].

All feature location techniques require a user-centered input on what to find. This can be done by picking some source code items from the system to be inspected such as variables or methods or domain language words to define the starting point for the analysis. In contrast to that, the COPG concept allows the user to define a generalized input based on object relations. Moreover, the COPG main target is to detect interacting objects whereby for any object all statements related to the object are collected. On the contrary, feature location techniques often cluster items of interest based on packages, methods, or classes and provide only a starting point for further code inspections.
11.4.5 Diagram Extraction

This section depicts approaches that reverse engineer diagrams based on the Unified Modeling Language (UML). The interaction among objects created by a program can be modeled with sequence or collaboration diagrams. Both interaction diagrams depict the exchange of messages between objects. Collaboration diagrams highlight the structural organization of the objects and sequence diagrams emphasize the time ordering of messages between objects. This section focuses on these two diagram types as they depict the interaction of several objects like the COPG does.

Interaction diagrams can be extracted either dynamically or statically. The dynamic extraction requires an executable system so that object creation and method invocation can be traced and extracted at runtime. Many approaches use this method to reverse engineer sequence diagrams out of legacy code [40, 41, 60, 94, 266, 298, 311, 355, 388, 389, 415].

There are also several approaches that reconstruct sequence diagrams based on static information. A comparison of several approaches has been presented by [290]. Moreover, Kollmann and Gogolla depict an approach based on code structures represented in a Java meta model [289]. Their algorithm walks through the meta model parts and creates collaboration diagrams by following predefined mapping rules.

Another algorithm for extracting object diagrams from source code has been presented by Tonella and Potrich [490]. They use an abstract program representation called Object Flow Graph (OFG). Such an OFG represents the created class instances and the related inter-object relationships. They extracted collaborations and sequence diagrams of the information given by the created OFGs [491]. Rountev and Connell use an interprocedural data-flow and object-naming analysis to extract sequence diagrams in their RED tool [407]. Later, they enhanced their work with an algorithm for mapping control-flow graphs to UML sequence diagrams [408]. The latest approach has been presented by Alvin et al. [17]. They use the source code to build a hypergraph and then extract sequence diagrams. Their StaticGen tool uses Soot and has been tested with several Android apps.

Besides the presented research approaches, several commercial and open-source tools statically generate sequence diagrams. Visual Paradigm generates sequence diagrams directly from source code in a one-to-one fashion [507]. The tools Visual Studio [337] and eUML2 Modeler [461] offer the user a possibility to refine diagrams by selecting methods of interest. An interactive construction method is offered by the Architexa tool [249].

In contrast to the COPG, sequence diagrams highlight the messages sent between objects. It focuses on the time sequence of message flows. Such a message flow is a method call of an object. This also applies to collaboration diagrams which emphasize the object organization. They embody numbered messages to depict the order of messages (method calls) between objects. Both diagram types only contain method calls; calls to other objects that have not been selected for the diagram are not shown. In contrast to that, our COPG approach considers all statements related to the selected interacting objects.
11.5 Static Security Analysis for Android

Software security is an emerging research area with a strong practical impact through the interconnectedness of software with the Internet. Hence, many approaches are dealing with static security analysis of software [80].

Static analysis is often used to detect common vulnerabilities such as buffer overflows or cross-site scripting in source code [168, 254]. Many approaches focus on implementation-level bugs which are mostly related to incorrect input validation [80, 310, 347, 517]. Some of these research prototypes have developed into commercial tools such as Fortify Source Code Analyzer [168] and Coverity Prevent [96]. Other approaches deal with programmer-written annotations for information flow that permit static code checks [344, 345, 412] or language-based security extensions to support the modeling of access information flow control [344].

Besides the various general purpose security analyses, many approaches have been tailored towards Android analysis needs. An overview of static analyses related to Android apps is given by Li et al. [306]. Since this work inspects the Android framework with regard to security patterns in Chapter 8 and uses Android applications for the case study in Chapter 9, the next sections briefly present approaches related to the used techniques within this thesis.

11.5.1 Modeling Android-Specific Concepts

Android apps provide special functional characteristics, which may constitute challenges for static analysis approaches as discussed in Section 9.4.
Many works in the area of Android analysis, however, only tackle some of the described analysis features [306]. Comparable approaches aiming at all of the static analysis challenges mentioned in Section 9.4 are [99, 248, 305, 338, 518, 536, 537]. Additionally, the many approaches deal with the problem of inter-app communication, where the communication between Android apps is considered [193, 287, 385, 442, 469]. This aspect is not in the focus of our approach.

Only some of the works mentioned before also consider Java-specific challenges to improve the results of the static Android app analysis. The works of Gordon et al., Huang et al., and Shen et al. model the Android-specific message mechanism for communicating through threads within an app [193, 248, 442]. This mechanism is called **Handler** and enables threads to communicate through a shared **Handler** object. Another Android extension to multi-threading is called **AsyncTask**. This extension provides a way to communicate with a user interface thread in an asynchronous fashion and is modeled in the works of Huang et al. and Shen et al. [248, 442]. Furthermore, Huang et al. model the implicit thread execution originating from Java, e.g., by inserting a call edge from the **start()** method of a **Thread** object to the object’s **run()** method [248].

Table 11.2 sums up the published approaches comparable to our proposed tool. None of the aforementioned approaches considers the fragment concept. Only Huang et al. [248] address all the aforementioned Android-specific programming concepts that we consider in our approach. We also simulate framework features related to Java concurrency concepts, such as **Runnable**, **ForkJoin**, **Callable**, and **ExecutorService**. In addition, our tool uses extracted **android.jar** files of running Android emulators [187] to have fully implemented API information instead of stubs, which are usually included in the **android.jar** distributed by the Android SDK. The extracted **android.jar** files enhance the generated points-to-analysis information for used API objects. None of the approaches listed in Table 11.2 describe whether they use extracted Android API information of the Android emulator instead of Android programming stubs for their analysis.

### 11.5.2 Slicing

Another often used technique among Android analyses is program slicing. Section 11.4.2 already depicted the general approaches of program slicing. With regard to Android, program slicing is often used for taint analyses and sometimes for vulnerability detection. In the following, we briefly describe some approaches of this research area.

**Taint Analysis**

Taint analyses inspect applications with regard to implemented (potentially) malicious data flows. These analyses can be conducted on different program representations such as bytecode. For example, the Apparecium tool works directly on disassembled bytecode of Android apps (Smali) and not on an abstract code representation such as Jimple [488]. They use forward and backward slices between
the marked source and sink for the taint analyses. Then, they combine the slices to eliminate paths that are only contained in one slice and do not end or start with the marked source or sink. AppCaulk is based on an early version of Apparecium [438]. This tool uses static taint analysis based on backward slicing to generate relevant points inside an application. These points are instrumented with a tainting logic to perform a dynamic taint analysis.

The SAAF (Static Android Analysis Framework for Android apps) tool provides program slicing for Smali code to perform data-flow analyses to backtrack parameters used by a given method [242]. It generates Smali files for all classes within an Android APK, parses the Smali files, and then creates an object representation of its contents based on the Manifest file, basic blocks of the methods, fields, and all opcodes.

Another tool that uses backward slicing for the taint analysis is the Brox tool [317]. It is based on the dalvik-opcode and uses a self-implemented data-flow analysis framework to detect Global Positioning System (GPS) information leakage in Android apps.

Besides the bytecode and Dalvik-opcode based approaches, some approaches use existing static analysis tools for their taint analysis approaches. Gibler et al.’s AndroidLeaks tool uses the WALA tool to perform taint-analyses with regard to permission information to detect leaked private information [175].

Two other tools compute slices on the internal representation of the Soot tool. Both tools convert the app’s application binary (DEX) format into Java bytecode and then translate the Java bytecode into an intermediate representation with Soot. Then, they use program slicing for the taint propagation. The Capper tool additionally rewrites (selectively) Android apps by inserting bytecode instructions for tracking sensitive information flows in specific fractions of the program that may be involved in information leakage [546]. In contrary, the tool AppSealer generates patches automatically for potential component hijacking attacks [547]. It injects a patch before a predefined policy is violated and shows a pop-up dialog to inform the user.

Besides the aim of taint analysis there are some other approaches using the slicing technique to identify vulnerabilities or security issues. The tool presented by Poeplau et al. uses slices to detect code loading sites for external code, such as class loading or native code, within Android apps [367]. They use an inter-procedural control-flow graph (ICFG) for their backward slicing to trace the flow of data starting at a selected instruction related to external code loading. The AQUA (Android Query Analyzer) tool uses static analysis to reverse engineer aspects of the application’s interaction with used databases [85]. It works directly with Android application binaries (DEX) and uses a lightweight program slicing technique. The slicing method extracts potentially relevant string information (nodes) that are used either directly or via content provider methods to construct SQLite queries.

Another tool that identified security issues via taint analysis is CredMiner [551]. It programmatically identifies and recovers (obfuscated) developer credentials which are unsafely embedded in Android apps. It locates the calls to interesting functions that can contain credential information via static backward slicing technique.
Vulnerability Detection

One work that can be directly compared to our tool is the CryptoLint tool [123]. CryptoLint finds implementation bugs in the cryptographic code of Android applications based on six rules (e.g., concerning insecure encryption modes, insecure parameters for password-based encryption, and weak random number generation) and was applied within a large-scale study. CryptoLint focuses on finding bugs at the level of single code lines as it uses cryptographic API calls and single parameters as criteria for backward slicing.

Other approaches deal with the wrong usage of the Android framework in apps. For example, CHEX allows an analyst to detect Android apps vulnerable to component hijacking, i.e., an attacker can jump on an app’s Android components and use them to access security-critical functionality [312]. CHEX uses static program analysis techniques based on system dependence graphs, which are usually employed to implement slicing. However, as CHEX does not use control dependences.

All in all, none of the presented approaches deal with the aspect of tracing multiple objects, which together constitute the implementation of a security feature (e.g., tracing the origin of the key and the data to be encrypted), rather than single parameters of crypto API calls [123]. The discussed taint analyses only provide path information based on data-flows from defined sources to sinks. In contrast to that, the approach in this thesis preserves live-time information of every object that interacts. Moreover, the tools are not designed to support locating security features in code and interactions between its constituting objects, which is one of the main tasks of our approach.

11.5.3 Android Analysis Evaluation

A well-known limitation of static analysis approaches is the false-positive and false-negative rate. This also applies to Android analysis tools, and proposed analysis techniques are evaluated in different ways. Due to the diverse approaches with regard to Android security analysis, we discuss only the tool evaluation of the comparable approaches described before and specifically focus on the evaluation with real-world applications. Table 11.3 sums up the discussed approaches.

Two works have not been evaluated against real-world apps. Klieber et al. [287] evaluate their tool against three self-written and three apps from the DroidBench benchmark suite [53] and Sufatrio et al. [469] evaluate against malware samples from the Android Malware Genome Project [550].

The tools SIG-Droid, BlueSeal, and DroidSafe have been evaluated by comparing the results to other analysis tools. Mirzaei et al. compare their tool SIG-Droid with two other approaches (Android Monkey and Dynodroid) [338]. They only use six open-source apps and compare the generated results. Gordon et al. compare their tool DroidSafe to FlowDroid [26] with analyzing 24 apps [193]. They assess the number of detected malicious flows of both tools. However, they do not attempt to analyze all of the remaining reported flows to determine whether they exist in the application or not and if they are malicious.
<table>
<thead>
<tr>
<th>Approach</th>
<th>Evaluated Real-World Apps</th>
<th>Evaluation Type</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>number of apps</td>
<td>SLOC</td>
</tr>
<tr>
<td></td>
<td></td>
<td>min</td>
</tr>
<tr>
<td>Cui et al. [99]</td>
<td>1,137</td>
<td>-</td>
</tr>
<tr>
<td>Gordon et al. [193]</td>
<td>24</td>
<td>200</td>
</tr>
<tr>
<td>Huang et al. [248]</td>
<td>182</td>
<td>-</td>
</tr>
<tr>
<td>Klieber et al. [287]</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>Li et al. [305]</td>
<td>15,000</td>
<td>276</td>
</tr>
<tr>
<td>Mirzaei et al. [338]</td>
<td>6</td>
<td>-</td>
</tr>
<tr>
<td>Ravitch et al. [385]</td>
<td>2,573</td>
<td>-</td>
</tr>
<tr>
<td>Shen et al. [442]</td>
<td>4,039</td>
<td>-</td>
</tr>
<tr>
<td>Sufatrio et al. [469]</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>Wei et al. [518]</td>
<td>853</td>
<td>-</td>
</tr>
<tr>
<td>Yang et al. [536]</td>
<td>20</td>
<td>2,380</td>
</tr>
<tr>
<td>Yang et al. [537]</td>
<td>20</td>
<td>2,380</td>
</tr>
</tbody>
</table>

Table 11.3 – Overview of evaluation strategies of static Android analysis tools. If a publication provides no data for an aspect it is marked with "-". Numbers in brackets "()" indicate the number of investigated apps if this differs from all evaluated apps.

Shen et al. validate their static analysis tool BlueSeal against 4,039 apps [442]. Due to problems with the used Soot framework, they could only analyze 2,885 apps. The results of the analyzed apps are then discussed. Moreover, they manually compare the generated flow permission of BlueSeal and TaintDroid [133] with 30 apps. TaintDroid performs a dynamic taint analysis for identifying malicious information flows. They also examine the intermediate representation of 100 apps to verify that BlueSeal detects flows, but they do not assess whether their tool has missed flows.

The remaining approaches listed in Table 11.3 are evaluated manually. Huang et al. use 182 different apps for their tool evaluation. The obtained results are discussed and 69 apps that have not been reported by their AsDroid tool are manually inspected to determine false negatives. Finally, they identify 11 apps missed by AsDroid, but do not give detailed information on their results.

Li et al., Cui et al., and Wei et al. analyze 753, 1,137, and 15,000, respectively, real-world apps. They investigate and discuss their results, but do not assess the recall of their tools by looking for missed leaks. This also applies to the 2,573 apps used for the evaluation of the FUSE tool by Ravitch et al. [385].

The developer team of the Gator tool applied their analysis on 20 open-source Android apps [536, 537]. They additionally perform a manual analysis on three apps to understand the sources of the tool imprecision [536]. Furthermore, the successor tool Gator3 has been evaluated with six apps in depth to determine the precision of the analysis [537].

All in all, the evaluation of Android analysis tools is often focused on inspecting the tool-generated results. Sometimes, tools are used for comparing the generated results when there are comparable and publicly available approaches for the
addressed achievement. Rarely, the whole app is used to determine the success of a tool and if so, such an evaluation is only conducted on a few apps. It is uncommon that a publication lists all analyzed apps with their size, which can be a hint on how complex the analyzed apps are. We have chosen the manual app evaluation to assess the precision and recall of our approach on 25 non-trivial real-world apps. The size per app is between 20,133 and 568,855 SLOC which is high compared to the collected size information in Table 11.3.
This chapter summarizes the achieved results and contributions of this thesis and proposes further research directions.

12.1 Summary and Conclusions

An important and established technique for designing software are patterns. They provide existing and time-proven solutions to a given problem. While the usage and consideration of security patterns is often focused on the design stage of software, we see, as missing work, that security patterns should also be considered in reviews and software maintenance activities.

The overall topic of this thesis is to validate published security patterns with regard to the needs of software maintenance and program comprehension activities. We believe that we have given more insight into the landscape, description, detection, and open issues of security patterns. In the following, the several contributions of this thesis are summarized in depth.

Collection of security patterns We have presented a collection of published security patterns in the period of 1997 to 2016. The collection list is up-to-date and contains more patterns than other security pattern collections [144, 238, 300, 437, 539]. Moreover, this thesis is a centralized location of security patterns, which can be beneficial to software designers and developers who need an overview of existing security patterns.

Classification of security patterns We have seen that existing security-pattern classifications are limited to organize only a few security patterns. In addition, most of them imply that the patterns to be organized have the software domain as target. However, we have also shown that many different security patterns exist and many of them cannot be considered as software-security patterns. Thus, we have introduced two new classification schemes and discussed challenges in classifying security patterns. The first classification scheme embraces 430 published security patterns and exceeds in numbers the existing
classifications by far. The second classification is based on 195 software-security patterns that we have obtained in the first organization process. It unites the focus of pattern recognition and security aspects. These classifications support software designers and analysts by the determined pattern characteristics.

**Analysis of software-security pattern descriptions**  The description forms of security patterns are very different. This heterogeneity is a problem for comparing security patterns. Therefore, the section definitions used in security-pattern templates have been collected and mapped to GoF and POSA design-pattern templates to identify additionally used sections.

In brevity our findings are: a) Most software-security patterns use a POSA-like description style; b) Their descriptions often have additional sections which are only used in a few different publications; c) These newly added sections are not security-specific, but the security potential of a security pattern can be stressed by using the *Forces* section.

Concluding, the heterogeneity of description forms is not essential to describe software-security patterns. The additionally used sections within security pattern descriptions can be mapped to the sections of POSA and GoF. Such a template does not need further significant sections to describe its security relevance or issue. We have shown that the developed mapping is an efficient way to compare the heterogeneously described security patterns and it can be used to consolidate pattern information. This technique can be used, for example, to compare different patterns or to unify pattern templates in case of the development of tool support.

**Clarification of differences between design and security patterns** We have presented the state-of-the-art of software-security patterns and discussed their maturity degree with regard to their classifications, description forms, code examples, and usage in the software life-cycle compared to the common design patterns.

In brevity, the results are: a) The area of pattern classification is well explored for security patterns; b) There is ongoing research and design pattern concepts have been adopted; c) The area of forward engineering seems to be well-investigated for security patterns; d) The description quality of software-security patterns needs to be improved; e) The documentation rate of UML diagrams in software-security patterns is low; f) Only one fourth of the software-security patterns provide code examples; g) The quality of examples is not as good as the design pattern examples; h) The research on security patterns is not so widespread in the area of software maintenance (design recovery, redocumentation, and software restructuring).

We see, as the main reason for this observation, the heterogeneous description nature of security patterns and the lack of proper code examples. We believe further investigations on the presentation quality and a unique description form of security patterns as well as an easy accessible repository presenting published patterns may improve the pattern’s reputation and their adoption.
Manifestation of security patterns in code We have collected security patterns that can be used to secure Android applications by mapping them to secure coding guidelines. For the three mostly mentioned security patterns from these guidelines we have distilled implementation variants based on the Android API. We have also depicted how diverse the object configuration among the variants of one security pattern is and described 27 in total. This pattern variant collection can be helpful for developers and analysts to obtain an overview of the several possibilities to implement a certain security feature in an Android application.

Detection of security patterns Only a few analysis tools take the well-known design patterns into account to support program comprehension at that point [503]. However, none of them considers the detection of security patterns. We have split the security-pattern detection in two parts.

In the first part we have used the hierarchical reflexion method implemented in the Bauhaus tool to detect security patterns at the architectural level. This work has highlighted possible issues on using security patterns for an (automatic) detection approach.

The second part presented how security patterns can be detected by their object interaction. This approach uses a low-level program representation to detect patterns with the help of COPGs, which are a further development of the OPG technique. To satisfy the need for detecting the interaction of many different objects that together form security patterns, we have enhanced the OPG concept to handle multiple objects. Our tool can be used for reviewing Android applications at later development stages when the security objectives are identified and a preliminary security scan prior to the software release is performed. It provides support for security audits by automatically detecting source code locations of implemented security features and tracing a security-relevant object’s configuration. Analysts can utilize the abstract representation of connected objects to understand security-object interactions.

The automatic pattern-variant detection has been evaluated in a case study with 25 non-trivial Android applications and shows acceptable results on precision and recall. In addition, this benchmark of benign Android apps can be used by researchers in other projects. Moreover, the automatic detection approach has been discussed with six software security experts from the SAFECode organization.

12.2 Future Research

This section presents and discusses collected items and ideas that are worth future research activities based on the presented topic.
12.2.1 Usage in Industry

By some (rare) studies (e.g., Hafiz et al. [208]) it has been shown that security patterns are used within the software development process. The list of collected software-security patterns in Section 4.4 is still long and some of them are tailored towards specific frameworks or contexts. We expect that not all patterns of this list are used to develop software. Therefore, it would be interesting to determine how many and which security patterns are of special interest for software development in industry. Imaginable are interviews with software security experts and/or conducting an online survey among security experts and industrial software developers.

Expert Interview

Expert interviews could be used to identify and prioritize software security patterns in matters of practical relevance and usage in commercial software systems. The interviewed experts can give insights into their procedure of assessing and analyzing software systems with regard to security. They can help to prioritize software security-patterns for further investigations by highlighting which security problems occur frequently and should be inspected more carefully than others. Moreover, it can be possibly clarified which role knowledge of software-security patterns plays within security audits and whether they are used to design commercial software systems.

Survey among Software Architects and Developers

Based on the previously described expert interview, a survey among software developers and architects in industry is conceivable. Here, developers could be interviewed directly with regard to the topic of security patterns. This could be of interest because many developers are not security experts. They have less knowledge on security, but they have to face the security topic during their work. Such a survey can provide insights into the attitude of a huge mass of developers towards developing secure systems. This survey could be exposed as an online survey to win many developers to participate. The goal of this survey is to identify which security patterns have practical relevance for developers. The frequently mentioned security patterns are of interest for further investigations in the area of designing, implementing, analyzing and reviewing software with regard to security.

12.2.2 Detection Improvement

This section presents several ways to improve the presented security pattern detection technique.

Security Pattern Variants

Section 8.1.2 presents a way to extract the shape of security patterns from API information. To overcome the discussed limitations, the *Known Use* section in a
security pattern description can be considered to obtain clues for suitable software systems that implement the security pattern. For example, the *Check Point* pattern is used at the login screen of Windows and Linux operating systems. Furthermore, software systems that provide the same (secured) functionality or service, e.g., FTP-Servers or mail servers can be inspected according to its security pattern usage. The main problem is here to find a particular security feature in the software. This can be done by reading the documentation (if available) or having an expert that evaluates the tools for implemented security features. Both ways can give clues for finding common used code snippets.

The collected variants can be compared to the API variants and "core" structures can be identified. These "core" structures can be used to implement a fuzzy search for security patterns in different systems and other pattern detection techniques can be used for their detection such as the described approaches in Section 11.2.1.

**Behavioral Analysis**

Enhancing static analysis with dynamic information is a powerful combination for analyses (see Chapter 11). The extraction of execution traces may be a good extension to the static program information of applications. Therefore, important behavioral aspects of security patterns must be identified, and the static analysis information base must be enhanced with dynamically collected information about the objects under surveillance. This step possibly enables one to additionally detect security patterns which rely more on behavior than object composition (Section 4.5.2). It is also conceivable to enhance techniques such as protocol recovery to distinct correctly from incorrectly implemented security patterns.

**Security Anti-Pattern Detection**

Besides design and security patterns, anti-patterns exist. They describe bad practices for a common problem [61]. Some prototypes exist which support the detection of anti-patterns in general [216, 253, 363]. They support in each case only special kinds of patterns like JEE or performance anti-patterns. The development of security anti-patterns that can detect wrong object combinations or problematic control-flow sequences within COPGs is conceivable. Therefore, the derived security pattern variants have to be inspected for possible insecure object configurations. The checking of control-flow sequences can be possibly done by enhancing existing protocol-recovery approaches such as the work presented by Quante and Koschke [375].

**12.2.3 Visualization**

The visualization of COPGs developed within the context of this thesis is quite basic. Thus, this section presents several improvements that can be made in visualizing security patterns to support analysts.
Usability aspects

The implemented COPG visualization could be enhanced. Features such as filtering certain nodes of interest, clustering nodes to groups, or views for certain aspects of the COPG are conceivable. For example, the Figure 10.2b in Section 10.2 displays a relatively small graph, but we have observed that COPGs can become large (up to 900 nodes) and hence quite confusing for an analyst. To use large COPGs effectively for reviews, one could display subviews of the COPGs, e.g., only call nodes could be shown or highlighted. In addition, a search feature for specific nodes or clusters of nodes can be implemented. For example, one can present a view for the encryption key or for the encryption mechanism.

Moreover, it could be beneficial to create different layout algorithms for COPGs whereby large graphs may be easier to browse. All these features may improve the usability of COPGs for program comprehension and should be further investigated by a controlled experiment.

Architectural View

The tool should also be able to create new views, comparable to views provided by the Bauhaus tool based on RFG information. These architectural views can be created containing only elements focusing on special purposes. It is conceivable to provide a view containing elements that are supposed to belong to a single security pattern. If one has identified more than one security pattern and created them on different views, one can create a new view by intersecting or uniting the view to visualize composed or merged security patterns as described by Schumacher [432]. Possibly this process can be automated when the tool has analyzed several security patterns. Presenting such combinations to analysts may facilitate the realization of adequately and inadequately programmed pattern collaborations. For instance, consider the combination of Single Access Point and Check Point pattern [437], where a badly implemented cooperation may raise security leaks.

Moreover, at architectural level security patterns can be marked as potential patterns and can be used to show deficiencies in the software architecture. With an automated check against the real source code, there can be detected architecture violations, such as calling a component not through a Check Point, that can induce security concept violations.

System View

With regard to the Android framework where several application can run aside, it would be nice to have an overview of a whole software system. Therefore, several analyzed applications could be combined in a single view at architectural level. Here, for example, in combination with taint analysis or other security analyses security bounds or layers of interacting applications could be modeled. With such a view data flows across applications and their implemented security provisions could be assessed more easily.
Conducting User Studies

The benefit of the developed tool and the aforementioned visualization should be evaluated by conducting controlled experiments. One experiment could answer whether COPGs are useful for program comprehension tasks. Another experiment could probably answer if the previously described visualization extensions are beneficial to understand the security architecture of software systems.

The implemented tool allows one to conduct user studies including think-aloud tests (recording a user’s comments on employing the tool) [398] and controlled experiments (one group with the tool, another one without) [373]. The challenge with controlled experiments is to define a test case that is complex enough such that the security-critical code is scattered across the application. At the same time, not too much training should be necessary to understand the test scenario. Edmundson et al. showed that analysts with more security experience did not necessarily perform better on reviewing applications with regard to security problems [120]. Thus, selecting a proper set of analysts for the experiment is another problem and can implicitly falsify the results.

Integrating the Tool into a Security Development Lifecycle (SDL)

In addition, the developed tool could be evaluated by embedding it in a security audit process such as SDL to point out possible benefits. The tool shall support the security analyst rather than replacing her. In particular, the tool is meant for enterprises who have established a well-defined SDL to support security and development teams. Since our technique connects architectural aspects to the concrete software implementation, the tool can be used in the early implementation phases within an SDL as well as after the implementation has been finished. The former application, however, is preferred because it gives the analyst/architect early or immediate feedback of whether security patterns are adequately used within the implementation.

12.3 Closing Words

The work in this thesis started with the intention to detect security patterns like Gamma’s design patterns. It turned out that the detection was not so easy due to the unique nature of the security patterns. Therefore, an extensive work on the security patterns themselves was conducted. Thereby, problems, opportunities for improvement, and possibilities for further research on them were discovered. Hopefully, this basis will be used to employ more applications to support the program comprehension task of software maintainers and reviewers to improve and ensure security in software systems.
BIBLIOGRAPHY


189


LIST OF FIGURES

1.1 Software engineering activities with some security touchpoints. . . . . . . . . . . . . 4
1.2 An analyst’s challenges on assessing an implemented security feature. . . . . . . . . 5
2.1 Topics related to the subject of this thesis. ................................................. 13
2.2 Software (re-)engineering activities with some security touchpoints. . . . . . . . . 14
3.1 Article discovery and inspection process. ................................................... 22
3.2 Article selection and discovery results. ..................................................... 24
3.3 This tag cloud depicts the amount of published publications per author. A larger
name indicates more, a smaller less publications per author. ............................ 26
3.4 Overview of research collaborations of security pattern publications. For the sake of
clarity authors who published publications on their own are left out. ................. 27
3.5 This word cloud depicts the multiply described patterns. The text size indicates
whether this pattern is described two times (small) or more often (high). .......... 28
4.1 Intuitive classification. ............................................................................. 30
4.2 GoF classification with a few examples [173]. ........................................... 31
4.3 Zimmer’s classification with a few examples [553]. .................................... 32
4.4 JEE pattern classification with a few examples [13]. ................................. 33
4.5 The classification by Konrad et al. with a few examples [291]. .................... 34
4.6 The classification by Steel et al. with a few examples [464]. ....................... 35
4.7 Design and security pattern classification relations – dashed lines between
publications highlight alike ideas. ................................................................. 38
4.8 Proceeding steps in our classification model. ............................................. 42
4.9 Application-domain based classification. ................................................... 43
4.10 Pattern distribution per domain. ............................................................. 44
4.11 Our software security-pattern classification with a few examples. .............. 46
4.12 Distribution of pattern-recognition and security aspects of organized software-
security patterns. The bubble size and number inside depicts how many security
patterns satisfy a certain aspect. ................................................................ 48
5.1 Collected sections in software-security publications. ............................... 53
5.2 Overview of POSA, GoF, and additional section usage. ............................. 55
5.3 Similarity analysis result for forms of security-pattern descriptions. ............ 56
5.4 UML class diagrams of the compared Authenticator patterns. .................... 65
6.1 The structure of Chapter 6. ................................................................. 71
6.2 The Singleton micro-architecture. .......................................................... 78
6.3 SLOC and number of commented lines in the inspected examples. ............. 82
7.1 Single Access Point UML diagrams (see also [437]). .................................. 89
7.2 Hypothetical architectures ..................................................................................... 90
7.3 Detection results for Spark application. ................................................................. 91
7.4 Detection results for Simple Android Instant Messaging Application. ................. 92

8.1 The pattern identification, authoring, and application process by Fehling et al. (see also [140]). Figure (a) depicts the whole process and Figure (b) highlights the activities of the pattern identification subprocess. ........................................ 97
8.2 Essential pattern sections and their writing order (see also [523]). ......................... 98
8.3 Micro-architecture extraction process. ..................................................................... 99
8.4 UML class diagram of the Authentication Enforcer pattern (see also [464]). .......... 102
8.5 UML class diagrams of the Authentication Enforcer variants A1 and A2. ............. 103
8.6 UML class diagram of the Authentication Enforcer variant A3. ............................. 103
8.7 UML class diagrams of the Authentication Enforcer variants A4 and A5. .......... 104
8.8 UML class diagrams of the Authentication Enforcer variants A6 and A7. ........... 105
8.9 UML class diagram of the variant A8. ................................................................. 106
8.10 UML class diagram of the Information Obscurity pattern (see also [437]). .......... 106
8.11 UML class diagram of the Information Obscurity variant I1. ............................... 107
8.12 UML class diagram of the Information Obscurity variant I2. ............................... 107
8.13 UML class diagram of the Information Obscurity variant I3. ............................... 108
8.14 UML class diagram of the Information Obscurity variant I4. ............................... 108
8.15 UML class diagram of the Information Obscurity variant I5. ............................... 108
8.16 UML class diagram of the Information Obscurity variant I6. ............................... 109
8.17 UML class diagram of the Information Obscurity variant I7. ............................... 109
8.18 UML class diagram of the Information Obscurity variant I8. ............................... 110
8.19 UML class diagram of the Secure Pipe pattern (see also [464]). ......................... 110
8.20 Variant S1 and S2 for creating a Secure Pipe. ...................................................... 111
8.21 Variant S3 and S4 for creating a Secure Pipe. ...................................................... 111
8.22 UML class diagram of the Secure Pipe variant S5. ............................................. 112
8.23 UML class diagrams of the Secure Pipe variants S6 and S7. ............................... 112
8.24 UML class diagrams of the Secure Pipe variants S8 and S9. .............................. 113
8.25 UML class diagrams of the Secure Pipe variants S10 and S11. ......................... 113

9.1 An analyst’s challenges on assessing an implemented security feature (see also Section 1.1). ............................................................................................................. 118
9.2 ICFG-based OPG construction example for the code from Listing 9.1. Figure (a) shows the ICFG, where the grey boxes indicate atomic nodes of the Cipher object. The crossed parts of the graph are removed to derive the OPG depicted in Figure (b). ................................................................................................................. 121
9.3 XML editor integrated in the Eclipse IDE. .......................................................... 125
9.4 COPG construction example based on OPGs. The [l. x] notation refers to the statement line in listings 9.1/9.2. Figure (a) shows extracted OPGs for the Cipher and the SecretKey object. The grey-marked boxes indicate a connection of the two graphs, which form a single connection node in the final COPG (Figure (b)). .... 126
9.5 Architecture of our analysis tool. ........................................................................... 127
9.6 Methods controlling the Android activity lifecycle [186]. .................................. 127
9.7 Overview of Activity and FragmentTransaction methods controlling the Fragment’s lifecycle. Fragment methods controlled via the Activity methods are highlighted with dashed arrows and the methods controlled via the FragmentTransaction object have solid arrows with numbers highlighting the call order. ............................................................. 131

10.1 A manually detected Authentication Enforcer variant in the SAP Mobile Documents app. ................................................................................................................... 143
10.2 Screenshots of a detected *Information Obscurity* COPG in the SIMATIC WinCC Sm@rtClient app. Blue backgrounded icons refer to a *Key* object and yellow ones to a *Cipher* object. Dashed circles mark creation and connection nodes in both figures. A double-click on a node allows the user to open the corresponding statement in the source code.

10.3 Top-level architecture view of SIMATIC WinCC Sm@rtClient app. Blue boxes indicate used security patterns (a dashed border indicates manually and a solid border tool-supported detected security patterns).

A.1 Used sections within security-pattern descriptions (1).

A.2 Used sections within security-pattern descriptions (2).
### LIST OF TABLES

1.1 Published and submitted papers covering content of this thesis. 10
1.2 Other publications not included in this thesis. 10
3.1 Conferences involved in the initial article selection. 22
4.1 Sections which are often used by security-pattern description forms. 40
5.1 An excerpt of description sections of three security pattern publications. 54
5.2 Mapping between design pattern sections [236]. 56
5.3 Two similar sections and their characteristics. 57
5.4 Alignment of assignable sections between POSA, GoF, and security patterns. The ⋆ character highlights a specialization of a section. 58
5.5 Alignment of mixture assignment sections between POSA, GoF, and security patterns. 59
5.6 Alignment of natively used sections between POSA, GoF, and security patterns. 59
5.7 Non assignable sections of security pattern publications. 60
5.8 Possible security-specific sections. 60
5.9 This table presents excluded multiply described patterns which are only described once or twice after removing patterns published by Yskout et al. [540]. Multiply described patterns depicted by Yskout et al. [540] are marked with a grey background. 62
5.10 This table lists all remaining patterns after applying the defined preselection rules, e.g., at least three descriptions of a pattern must exist. For a better overview, multiply described patterns by Yskout et al. [540], Blakley et al. [50], and Fernandez [144] are marked with a grey background as excluded pattern descriptions. 63
5.11 The pattern comparison setup. A grey background marks the usage of the section alignment for the comparison process. 63
5.12 Time measuring results. A grey background marks the usage of the section alignment for the comparison process. 64
6.1 Additional documentation type usage. 79
6.2 Results of code example inspection for security patterns. 80
6.3 Results of the C++ code example inspection for security patterns. 81
8.1 Collected guidelines discussing secure (Android) application development. 100
8.2 Number of extracted security pattern variants. 114
9.1 Variants using the same object configuration. 134
9.2 Number of security pattern variants used for tool testing. 134
10.1 Excluded Android applications due to identified problematic factors. 138
<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.2</td>
<td>Selected Android apps for the case study with their version number and category in Google’s Play market (B=Business, C=Communication, H=Health &amp; Fitness, L=Lifestyle, M=Maps &amp; Navigation, P=Productivity, S=Social, T=Tools).</td>
</tr>
<tr>
<td>10.3</td>
<td>Our case study results per analyzed app with the number of instances found by manual review and our tool as well as the number of true instances.</td>
</tr>
<tr>
<td>11.1</td>
<td>Overview of venues collecting security patterns.</td>
</tr>
<tr>
<td>11.2</td>
<td>Overview of approaches providing a comparable modeling of Android framework specialties.</td>
</tr>
<tr>
<td>11.3</td>
<td>Overview of evaluation strategies of static Android analysis tools. If a publication provides no data for an aspect it is marked with &quot;-&quot;. Numbers in brackets &quot;(())&quot; indicate the number of investigated apps if this differs from all evaluated apps.</td>
</tr>
<tr>
<td>B.1</td>
<td>Organized security patterns in the application domains cryptographic, enterprise, network, and user (1).</td>
</tr>
<tr>
<td>B.2</td>
<td>Organized security patterns in the application domains cryptographic, enterprise, network, and user (2).</td>
</tr>
<tr>
<td>B.3</td>
<td>Organized security patterns in the application domains cryptographic, enterprise, network, and user (3).</td>
</tr>
<tr>
<td>B.4</td>
<td>Organized security patterns in the application domains cryptographic, enterprise, network, and user (4).</td>
</tr>
<tr>
<td>B.5</td>
<td>Organized security patterns in the application domains cryptographic, enterprise, network, and user (5).</td>
</tr>
<tr>
<td>B.6</td>
<td>Organized security patterns in the application domains cryptographic, enterprise, network, and user (6).</td>
</tr>
<tr>
<td>C.1</td>
<td>Software-security patterns classified by security aspects and recognition needs (1).</td>
</tr>
<tr>
<td>C.2</td>
<td>Software-security patterns classified by security aspects and recognition needs (2).</td>
</tr>
<tr>
<td>C.3</td>
<td>Software-security patterns classified by security aspects and recognition needs (3).</td>
</tr>
<tr>
<td>C.4</td>
<td>Software-security patterns classified by security aspects and recognition needs (4).</td>
</tr>
<tr>
<td>C.5</td>
<td>Software-security patterns classified by security aspects and recognition needs (5).</td>
</tr>
<tr>
<td>C.6</td>
<td>Software-security patterns classified by security aspects and recognition needs (6).</td>
</tr>
<tr>
<td>C.7</td>
<td>Software-security patterns classified by security aspects and recognition needs (7).</td>
</tr>
<tr>
<td>E.1</td>
<td>Collected guideline information mapped to security patterns (1).</td>
</tr>
<tr>
<td>E.2</td>
<td>Collected guideline information mapped to security patterns (2).</td>
</tr>
</tbody>
</table>
LIST OF CODE EXAMPLES

9.1 Encryption example. ................................................................. 119
9.2 Encryption example help file. ...................................................... 119
9.3 COPG input example. ................................................................. 124
9.4 XML-based COPG input. .............................................................. 125
9.5 Android ICC example. ................................................................. 128
9.6 Android UI callback example. ....................................................... 128
9.7 Android AsyncTask example. ......................................................... 129
9.8 Android XML layout example. ....................................................... 129
9.9 Android layout example. .............................................................. 129
9.10 Embedding a Fragment statically by inflating an XML layout file. .... 132
9.11 XML layout file using a fragment. ................................................ 132
9.12 A Fragment example class. .......................................................... 132
9.13 XML layout file using a placeholder for dynamically processed Fragments. .. 132
9.14 An example for adding a Fragment dynamically. ............................. 132
9.15 Runnable example ................................................................. 133
GLOSSARY

Access Control  The security concept access control determines which principal (e.g., user, machine, and process) may access which data.

Accountability  The security concept accountability specifies that certain actions are logged for audits.

Authentication  Authentication is a process or action of verifying principals (e.g., users, machines, and processes).

Availability  The concept availability means that data/services are accessible when needed.

Bug  A bug is a (security) malfunction of a part of system which produces undesirable result such as buffer overflows that stem from erroneous coding.

Confidentiality  With regard to security, confidentiality guarantees the secrecy of data.

Flaw  A flaw with regard to security refers to a security problem that is manifested in the implementation but may stem from the design.

Integrity  The security concept integrity describes that data are not modified in an unauthorized way.

Jimple  Jimple is a typed 3-address intermediate representation built in the Soot tool.

Micro-Architecture  A micro-architecture is a composition of classes, methods, and fields; their structure and organization express an occurrence of a design motif in code.

Non-Repudiation  The security concept non-repudiation describes that an action can be proved and cannot be denied later.

Pattern Mining  Pattern mining is the discovery process of patterns contained in our minds or in activities associated with a certain target.

Taint Analysis  Taint analyses inspect applications with regard to implemented (potentially) malicious data flows.

UMLsec  UMLsec extends the Unified Modeling Language (UML) and integrates security-related information into UML specifications [528].

ZIP  ZIP is an archive file format that supports lossless data compression.
ACRONYMS

ACM  Association for Computing Machinery.
API  Application Programming Interface.
APK  Android Package Kit.
ASG  Abstract Semantic Graph.
AsianPLoP  Asian Conference on Pattern Languages of Programs.
AST  Abstract Syntax Tree.
BSI  Bundesamt für Sicherheit in der Informationstechnik (engl. German Federal Office for Information Security).
CAPEC  Common Attack Pattern Enumeration and Classification.
CIA  Confidentiality, Integrity, and Availability.
COPG  Connected Object Process Graph.
CWE  Common Weakness Enumeration.
DEX  Dalvik Executable format.
DEXA  International Conference on Database and Expert Systems Applications.
DFD  Data Flow Diagram.
DOM  Document Object Model.
DOPG  Dynamic Object Process Graph.
DPML  Design Pattern Markup Language.
EuroPLoP  European Pattern Languages of Programs Conference.
FCA  Formal Concept Analysis.
FTP  File Transfer Protocol.
GoF  Gang-of-Four.
GPS  Global Positioning System.
HTML HyperText Markup Language.
HTTP Hypertext Transfer Protocol.
HTTPS Hypertext Transfer Protocol Secure.
IAC Inter-App Communication.
ICC Inter-Component Communication.
ICFG inter-procedural control-flow graph.
ICOMP International Conference on Internet Computing.
IDE Integrated Development Environment.
IEEE Institute of Electrical and Electronics Engineers.
IML Intermediate Language.
IR Information Retrieval.
IT Information Technology.
JSP JavaServer Pages.
KoalaPLoP Australian Conference on Pattern Languages of Programs.
NLP Natural Language Processing.
OFG Object Flow Graph.
OPG Object Process Graph.
OWASP Open Web Application Security Project.
PATTERNS International Conferences on Pervasive Patterns and Applications.
PC Personal Computer.
PDG Program Dependence Graph.
PLML Pattern Language Markup Language.
PLoP Pattern Languages of Programs Conference.
POSA Pattern-Oriented Software Architecture.
PPTP Point-to-Point Tunneling Protocol.
RBAC Role-based Access Control.
RFG Resource Flow Graph.
SDG System Dependence Graph.
SDK Software Development Kit.
SDL Security Development Lifecycle.
Appendix

**SEKE**  International Conference on Software Engineering and Knowledge Engineering.

**SLOC**  Source Lines of Code.

**SOA**  Service-Oriented Architecture.

**SSL**  Secure Sockets Layer.

**STRIDE**  Spoofing, Tampering, Repudiation, Information disclosure, Denial of service, and Elevation of privilege.

**SugarLoafPloP**  American Conference on Pattern Languages of Programming.

**TLS**  Transport Layer Security.

**UI**  User Interface.

**UML**  Unified Modeling Language.

**URL**  Uniform Resource Locator.

**VikingPloP**  The Nordic Conference on Pattern Languages of Programs.

**XML**  Extensible Markup Language.
The collected sections of the security-pattern publications in Section 4.3.2 and Section 5.2.1 are presented in the figures A.1 and A.2.

Figure A.1 – Used sections within security-pattern descriptions (1).
### Figure A.2 – Used sections within security-pattern descriptions (2).
The tables B.1 to B.6 depict the security patterns organized in the application domains cryptographic, enterprise, network, and user according to Section 4.4.2. Security patterns belonging to the software domain are given in Appendix C. If patterns have been described by more than one publication, all publications describing the pattern are given in chronological order.

<table>
<thead>
<tr>
<th>Security Pattern Name</th>
<th>Cryptographic</th>
<th>Enterprise</th>
<th>Network</th>
<th>User</th>
</tr>
</thead>
<tbody>
<tr>
<td>WS-Reliability [63]</td>
<td></td>
<td></td>
<td></td>
<td>●</td>
</tr>
<tr>
<td>WS-Reliable Messaging [63]</td>
<td></td>
<td></td>
<td></td>
<td>●</td>
</tr>
<tr>
<td>3rd Party Communication [401]</td>
<td></td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Abstract IDB [144]</td>
<td></td>
<td></td>
<td>●</td>
<td></td>
</tr>
<tr>
<td>Abstract Virtual Private Network [144]</td>
<td></td>
<td></td>
<td>●</td>
<td></td>
</tr>
<tr>
<td>Abuse [357]</td>
<td></td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Acceptable Uncertainty [269]</td>
<td></td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Access Control Requirements [437]</td>
<td></td>
<td></td>
<td>●</td>
<td></td>
</tr>
<tr>
<td>Access Control to Physical Structures [155]</td>
<td></td>
<td>●</td>
<td></td>
<td>●</td>
</tr>
<tr>
<td>Account Category [391]</td>
<td></td>
<td>●</td>
<td></td>
<td>●</td>
</tr>
<tr>
<td>Actor-based Access Rights [130]</td>
<td></td>
<td></td>
<td>●</td>
<td></td>
</tr>
<tr>
<td>Address Book [302]</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Address Filter Firewall [153]</td>
<td></td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Alarm Monitoring [155]</td>
<td></td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Alice and Friends [302]</td>
<td></td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Anonymity Set [204]</td>
<td></td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Application Firewall [107, 144, 540]</td>
<td></td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Application Proxy Firewall [153]</td>
<td></td>
<td>●</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table B.1 – Organized security patterns in the application domains cryptographic, enterprise, network, and user (1).
<table>
<thead>
<tr>
<th>Security Pattern Name</th>
<th>Cryptographic</th>
<th>Enterprise</th>
<th>Network</th>
<th>User</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architectural Solution Conformance [136]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Architectural Solution Definition [136]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Architectural Solution and Technology Mapping [136]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Asset Evaluation [437]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Asymmetric Encryption [144]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Audit Requirements [437]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Audit Trails and Logging Requirements [437]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Automated I and A Design Alternatives [437]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Batched Routing [204]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Behavior-Based IDS [144]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Biometrics Design Alternatives [437]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Business Application Planning Profile [136]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Business Application and Organizational Unit Cluster Map [136]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Business Application and Organizational Unit Relationship [136]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Certificate Authority [303]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Certificate Revocation [303]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Chinese Wall [101]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Code Book [391]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Comparator Checked Fault Tolerant System [50, 540]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Confidential Data Being Sent [357]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Constant Length Padding [204]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Constant Name Padding [204]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cover Traffic [204]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cryptographic Key Generation [303]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cryptographic Metapattern [59]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cryptographic Protocol [436]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cryptographic Smart Card [303]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data Decryption [97]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data Integrity in P2P Systems [429]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Defence in Depth [132]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Delayed Routing [204]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Deterministic Search for Witness [269]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dictionary Word [391]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Digital Signature with Hashing [144, 226]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Divide and Conquer [132]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dynamically Adjustable Non-Functional Configuration [117]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>End to End Security [132]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Enlist the Users [132]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Enterprise Partner Communication [402, 437]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Enterprise Security Approaches [437]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Enterprise Security Services [437]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Entity to Entity Security [132]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Error Detection And Correction [50]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table B.2 – Organized security patterns in the application domains cryptographic, enterprise, network, and user (2).
<table>
<thead>
<tr>
<th>Security Pattern Name</th>
<th>Cryptographic</th>
<th>Enterprise</th>
<th>Network</th>
<th>User</th>
</tr>
</thead>
<tbody>
<tr>
<td>Face-to-face [302]</td>
<td></td>
<td></td>
<td></td>
<td>●</td>
</tr>
<tr>
<td>Firewall [435, 540]</td>
<td></td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fortress Mentality [132]</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hidden Metadata [204]</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>I and A Requirements [437]</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IPSec VPN [144]</td>
<td></td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ignore Security Incidents [132]</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ignore Security Patches [132]</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Immune System [132]</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Increased Monitoring for Intellectual Property (IP) Theft by Departing Insiders [339]</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Information Secrecy [59]</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Informed Consent for Web-Based Transactions [404]</td>
<td></td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Integral Data Being Received [357]</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Integral Data Being Sent [357]</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Integration Reverse Proxy [437]</td>
<td></td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intrusion Detection Requirements [437]</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Issue Driven [132]</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Just do it Together [132]</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Keep it Open [132]</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Keep it Secret [391]</td>
<td></td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Keep it Simple [132]</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Key in the Pocket [302]</td>
<td></td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lay it Open [391]</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Layered Encryption [204]</td>
<td></td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Layered Security [132, 401]</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Low Hanging Fruit [401, 403]</td>
<td></td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Make it Complex [132]</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Manage Risk [132]</td>
<td></td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Masking Your Online Traffic [404]</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Master Account File [391]</td>
<td></td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mature Through Time [132]</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Message Authentication [59]</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Message Integrity [59]</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Minimal Information Asymetry [404]</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Morphed Representation [204]</td>
<td></td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mutal Key PoK [98]</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Need to Know [132]</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Need to Protect [132]</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Network Authentication Protocol [436]</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Network Encryption Protocol [436]</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Network Functions Virtualization [145]</td>
<td></td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Non-Repudiation Requirements [437]</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Obey the Law [132]</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Object Filter and Access Control Framework [234]</td>
<td></td>
<td>●</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table B.3** – Organized security patterns in the application domains cryptographic, enterprise, network, and user (3).
<table>
<thead>
<tr>
<th>Security Pattern Name</th>
<th>Cryptographic</th>
<th>Enterprise</th>
<th>Network</th>
<th>User</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operational Monitoring and Alerting [117]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Packet Filter [435]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Packet Filter Firewall [437]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Paralysis by Analysis [132]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Password Algorithm [391]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Password Design and Use [437]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Password Externalization [391]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Password Hint [391]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Password Lock Box [391]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Password Salt [391]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Password synchronizer [391]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Peace of War [132]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Perimeter Defence [132]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Point Solutions [132]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Privacy-Aware Network Client Pattern [413]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Private Data Being Received [357]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Proactive Governance [132]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Property Certification [98]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Protection Against Cookies [433]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Protection Reverse Proxy [437]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Proxy [435]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Proxy-Based Firewall [437]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pseudonymous Email [433]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Public Key Database [303]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Public Key Exchange [303]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Random Pick [269]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Relays [155]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reliable Security [64]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Remote Authenticator/Authorizer [144, 152]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Replicated System [50, 540]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Respond on Security Incidents [132]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Return on Investment [132]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reverse Proxy [457]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Risk Assessment and Management [401, 402]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Risk Avoidance [132]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Risk Determination [437]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Risk Unawareness [132]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Role Rights Definition [437]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Role-Based Security Assertion Coordinator [143]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SAML-Assertion [144]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Safety Before Security [132]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Safety Unawareness [132]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Seal Ring Engraver [302]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sealed Envelope [302]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sealed and Signed Envelope [302]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Table B.4 – Organized security patterns in the application domains cryptographic, enterprise, network, and user (4).*
<table>
<thead>
<tr>
<th>Security Pattern Name</th>
<th>Cryptographic</th>
<th>Enterprise</th>
<th>Network</th>
<th>User</th>
</tr>
</thead>
<tbody>
<tr>
<td>Secrecy with Appendix [59]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Secrecy with Appendix and Signature [59]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Secrecy with Authentication [59]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Secrecy with Integrity [59]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Secrecy with Signature [59]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Secure Communications [303]</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Secure Domain Name System (SDNS) [146]</td>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Secure Match [98]</td>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Secure Proxy [50]</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Secure Reliability [64]</td>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Secure and Safe Port Loading Facility [162]</td>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Security Accounting Requirements [437]</td>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Security Goals Before Means [132]</td>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Security Guard [132]</td>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Security Needs Identification for Enterprise Assets [437]</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Security Reverse Proxy [540]</td>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Security as a Business Issue [132]</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Security as a Desert [132]</td>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Security as a Technical Issue [132]</td>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Security at Any Price [132]</td>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Security in Every Change [132]</td>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Security to Obscurity [132]</td>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Sender Authentication [59]</td>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Sensor Encryption [97]</td>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Session Key Exchange With Client and Server-Side Certificates [303]</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Session Key Exchange With Server-Side Certificates [303]</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Session Key Exchange with Public Eyes [303]</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Session Management [357]</td>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Shared Data Being Received [357]</td>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Signature with Appendix [59]</td>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Signature-Based IDS [144]</td>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Signed Envelope [302]</td>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Singleton Password [391]</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Standard Conformity Management [136]</td>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Standards Conformity Exceptions [136]</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Standby [50]</td>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Stateful Firewall [437]</td>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Stay Current and Ahead [391]</td>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Swift Sieve [269]</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Symmetric Encryption [144, 225]</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Symmetric Key Cryptography [450]</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>System Overview [117]</td>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>TLS Virtual Private Network [144]</td>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Technology Usage [136]</td>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
</tbody>
</table>

**Table B.5** – Organized security patterns in the application domains cryptographic, enterprise, network, and user (5).
### Security Pattern Name

<table>
<thead>
<tr>
<th>Pattern Name</th>
<th>Cryptographic</th>
<th>Enterprise</th>
<th>Network</th>
<th>User</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Effect of Safeguards [132]</td>
<td></td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>The Forged Seal Ring [302]</td>
<td></td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>The Goal of Security [132]</td>
<td></td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>The Nature of Safeguards [132]</td>
<td></td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>The Network as a Battleground [132]</td>
<td></td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>The Real Thing [302]</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>The Security Provider [402]</td>
<td></td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>There is Somebody Eavesdropping [302]</td>
<td></td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Threat Assessment [437]</td>
<td></td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Three-Point Logging [117]</td>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Time Based Security [132]</td>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Top Down Approach [132]</td>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Transport Layer Security (TLS) [144, 297]</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Trust Nobody [132]</td>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Trust Your Employees [132]</td>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Trust Your Security [132]</td>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Trust Your Vendor [132]</td>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Typing Algorithm [391]</td>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Uncontrolled Access [132]</td>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Unusual Variation [391]</td>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Violate the Law [132]</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Virtual Private Network [296]</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Virtual Private Networks [436]</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Vulnerability Assessment [437]</td>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>WS-Policy [6, 144]</td>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>WS-SecureConversation [7]</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>WS-Security [144, 227]</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>WS-Trust [5, 144]</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Wait for the Auditor [132]</td>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Watch the Watchers [132]</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Web Service Policy Language (WSPL) [105, 144]</td>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>White Hat, Hack Thyself [403]</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>White Hat, Hack Thyself [401]</td>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Whitelisting Firewall (WLF) [263]</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>WiMax Security [157]</td>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Winmax Network Architecture [157]</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>XACML Access Control Evaluation [105, 144]</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>XACML Authorization [105, 144]</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>XML Encryption [144, 225]</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>XML Firewall [107, 143, 144]</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>XML Signature [144, 226]</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
</tbody>
</table>

Table B.6 – Organized security patterns in the application domains cryptographic, enterprise, network, and user (6).
SECURITY-PATTERN CLASSIFICATION: SECURITY ASPECTS AND RECOGNITION NEEDS

The tables C.1 to C.7 depict the classification of all software-security patterns that were collected during the literature review process of Chapter 4. The magnifier icon (%) highlights to which “Matching Aspect” a pattern belongs, and the lock icon (🔒) shows which security aspects a pattern addresses. Some patterns have been described by more than one publication. Therefore, all publications describing a pattern are given in chronological order.

<table>
<thead>
<tr>
<th>Pattern Name</th>
<th>Matching Aspects</th>
<th>Security Aspects</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Structural</td>
<td>Access Control</td>
</tr>
<tr>
<td></td>
<td>Behavioral</td>
<td>Authentication</td>
</tr>
<tr>
<td></td>
<td>Generic Concept</td>
<td>Accountability</td>
</tr>
<tr>
<td></td>
<td>Access Control</td>
<td>Availability</td>
</tr>
<tr>
<td></td>
<td>Behavioral</td>
<td>Confidentiality</td>
</tr>
<tr>
<td></td>
<td>Generic Concept</td>
<td>Integrity</td>
</tr>
<tr>
<td></td>
<td>Structural</td>
<td>Non-Repudiation</td>
</tr>
<tr>
<td>Access Control List [106, 144]</td>
<td>📷</td>
<td>🛠</td>
</tr>
<tr>
<td>Access Controller [342]</td>
<td>📷</td>
<td>🛠</td>
</tr>
<tr>
<td>Access Session [149]</td>
<td>📷</td>
<td>🛠</td>
</tr>
<tr>
<td>Account Lockout [282]</td>
<td>📷</td>
<td>🛠</td>
</tr>
<tr>
<td>Actor and Role Lifecycle [130]</td>
<td>📷</td>
<td>🛠</td>
</tr>
<tr>
<td>Administrator Hierarchy [144, 154]</td>
<td>📷</td>
<td>🛠</td>
</tr>
<tr>
<td>Administrator Objects [288]</td>
<td>📷</td>
<td>🛠</td>
</tr>
<tr>
<td>Agency Guard [342]</td>
<td>📷</td>
<td>🛠</td>
</tr>
<tr>
<td>Agent Authenticator [342]</td>
<td>📷</td>
<td>🛠</td>
</tr>
<tr>
<td>Assertion Builder [464]</td>
<td>📷</td>
<td>🛠</td>
</tr>
</tbody>
</table>

Table C.1 – Software-security patterns classified by security aspects and recognition needs (1).
<table>
<thead>
<tr>
<th>Pattern Name</th>
<th>Matching Aspects</th>
<th>Security Aspects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Audit Interceptor [464, 540]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Authenticated Session [282]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Authentication Enforcer [464, 540]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Authenticator [50, 144, 150, 270, 437]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Authoritative Source of Data [401, 402]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Authorization [144, 151, 437]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Authorization Enforcer [464, 540]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Build the Server from the Ground Up [282]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Capability [106, 144]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Check Point [437, 538]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Checkpointed System [50, 208, 540]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Choose the Right Stuff [282]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Circle of Trust [144]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Clear Sensitive Information [115]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Client Data Storage [282]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Client Input Filters [282]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cloud Policy Management Point [121]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Compartemenzation [208]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Container Managed Security [464, 540]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Content Independent Processing [208]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Content-dependent Authorizer [146]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Context-enhanced Authorizer [146]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Controlled Access Session [144]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Controlled Execution Domain [144]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Controlled Execution Environment [142, 437]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Controlled Virtual Address Space [437]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Controlled-Object Factory [144, 150, 437, 540]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Controlled-Object Monitor [144, 150, 437, 540]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Controlled-Process Creator [144, 150, 437, 540]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Credential [144, 341]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Credential Delegation [522]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Credential Tokenizer [464, 540]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Defer to Kernel [115]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Demilitarized Zone [437, 540]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table C.2 – Software-security patterns classified by security aspects and recognition needs (2).
## Appendix C

### Matching Aspects

<table>
<thead>
<tr>
<th>Pattern Name</th>
<th>Structural</th>
<th>Behavioral</th>
<th>Generic Concept</th>
<th>Access Control</th>
<th>Accountability</th>
<th>Availability</th>
<th>Confidentiality</th>
<th>Integrity</th>
<th>Non-Repudiation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Directed Session [282]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Distributed Responsibility [208]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Distrustful Decomposition [115]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Document the Security [282]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Document the Server Configuration [282]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dynamic Integrity Properties [384]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dynamic Service Management [464]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Encrypted Storage [282]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Enroll Using Third-Part Validation [282]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Enroll Without Validating [282]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Enroll by Validating out of Band [282]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Enroll with a Pre-Existing Shared Secret [282]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Execution Domain [142, 144, 437]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fail Securely [132, 401, 403]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fault Container [424]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>File Access Control [144]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>File Authorization [142, 151, 437]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fine-grain Access Control [409]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Front Door [437]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Full Access With Errors [437]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Full View with Errors [538, 540]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Goals [282]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Grant-Based Access Control Pattern (GBAC) [97]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hidden Implementation [282]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>History-based Authentication [87]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hot Standby [208]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ID/Password Authentication [357]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Identity Federation [144]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Identity Provider [144]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Incepting Web Agent [464]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Information Obscurity [437]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Input Guard [424, 540]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Input Validation [115]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table C.3** – Software-security patterns classified by security aspects and recognition needs (3).
<table>
<thead>
<tr>
<th>Pattern Name</th>
<th>Matching Aspects</th>
<th>Security Aspects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integrity Attestation (Remote Attestation) [383]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Integrity Protection [383]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intercepting Validator [464]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Keep Session Data in Client [475, 475]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Keep Session Data in Server [475]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Known Partners [437]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Layered Operating System Architecture [144]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Least Privilege [161]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Liberty Alliance Identity Federation [144]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Limited Access [437]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Limited View [538, 540]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load Balancer [475, 540]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Log For Audit [282]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Message Inspector [464]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Message Interceptor Gateway [464]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Microkernel Operating System Architecture [144]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Minefield [282]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Modular Operating System Architecture [144]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multilevel Secure Partitions [147]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multilevel Security [144, 151, 437]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiple Secure Observers Using J2EE [180]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Network Address Blacklist [282]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Obfuscated Transfer Object [464, 540]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Output Guard [424, 540]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Partitioned Application [282]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Password Authentication [282]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Password Based Key Exchange [450]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Password Propagation [282]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Password Synchronizer [464]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Patch Proactively [282]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pathname Canonicalization [115]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Policy [50]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Policy Delegate [464]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Policy Enforcement [552]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table C.4** – Software-security patterns classified by security aspects and recognition needs (4).
<table>
<thead>
<tr>
<th>Pattern Name</th>
<th>Matching Aspects</th>
<th>Security Aspects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Policy-Based Access Control [106, 144]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PrivSep (Privilege Separation) [115]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Privilege-limited Role [288]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Protected Entry Point [144]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Protected Entry Points [147]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Protected System [50]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Protection Rings [144, 147]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Recoverable Component [208]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Red Team the Design [282]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reference Monitor [142, 437]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reified Reference Monitor [144]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Resource Acquisition Is Initialization (RAII) [115]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Right Based Fine Grain Access Control [409]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Risk-Based Authenticator [465]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Role Based Fine Grain Access Control [409]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Role Validator [288]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Role-Based Access [288]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Role-Based Access Control (RBAC) [144, 151, 437]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Role-Hierarchies [288]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Roles [538]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Safe Data Structure [208]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sandbox [342]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Secret Partner [496]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Secure Access Layer [538, 540]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Secure Adapter [144]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Secure Assertion [282]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Secure Based Action [464]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Secure Blackboard [144, 360]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Secure Broker [144, 340]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Secure Builder Factory [115]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Secure Chain of Responsibility [115]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Secure Channels [437]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table C.5 – Software-security patterns classified by security aspects and recognition needs (5).
<table>
<thead>
<tr>
<th>Pattern Name</th>
<th>Matching Aspects</th>
<th>Security Aspects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Secure Communication [50]</td>
<td>📌</td>
<td>🔒</td>
</tr>
<tr>
<td>Secure Directory [115]</td>
<td>📌</td>
<td>🔒</td>
</tr>
<tr>
<td>Secure Distributed Publish/Subscribe [144]</td>
<td>📌</td>
<td>🔒</td>
</tr>
<tr>
<td>Secure Enterprise Service Bus [144]</td>
<td>📌</td>
<td>🔒</td>
</tr>
<tr>
<td>Secure Factory [115]</td>
<td>📌</td>
<td>🔒</td>
</tr>
<tr>
<td>Secure Logger [115, 464, 540]</td>
<td>📌</td>
<td>🔒</td>
</tr>
<tr>
<td>Secure Mediator [264]</td>
<td>📌</td>
<td>🔒</td>
</tr>
<tr>
<td>Secure Message Router [464, 540]</td>
<td>📌</td>
<td>🔒</td>
</tr>
<tr>
<td>Secure Model-View-Controller [144]</td>
<td>📌</td>
<td>🔒</td>
</tr>
<tr>
<td>Secure Pipe [464, 540]</td>
<td>📌</td>
<td>🔒</td>
</tr>
<tr>
<td>Secure Pipes and Filters [144, 148]</td>
<td>📌</td>
<td>🔒</td>
</tr>
<tr>
<td>Secure Preforking [203]</td>
<td>📌</td>
<td>🔒</td>
</tr>
<tr>
<td>Secure Process/Thread [144, 154]</td>
<td>📌</td>
<td>🔒</td>
</tr>
<tr>
<td>Secure Service Facade [464, 540]</td>
<td>📌</td>
<td>🔒</td>
</tr>
<tr>
<td>Secure Service Proxy [464]</td>
<td>📌</td>
<td>🔒</td>
</tr>
<tr>
<td>Secure Session Object [464, 540]</td>
<td>📌</td>
<td>🔒</td>
</tr>
<tr>
<td>Secure Singleton [264]</td>
<td>📌</td>
<td>🔒</td>
</tr>
<tr>
<td>Secure State Machine [115]</td>
<td>📌</td>
<td>🔒</td>
</tr>
<tr>
<td>Secure Strategy Factory [115]</td>
<td>📌</td>
<td>🔒</td>
</tr>
<tr>
<td>Secure Three-Tier Architecture [144]</td>
<td>📌</td>
<td>🔒</td>
</tr>
<tr>
<td>Secure Virtual Machine Image Repository [121]</td>
<td>📌</td>
<td>🔒</td>
</tr>
<tr>
<td>Secure Visitor [115]</td>
<td>📌</td>
<td>🔒</td>
</tr>
<tr>
<td>Security Association [50, 540]</td>
<td>📌</td>
<td>🔒</td>
</tr>
<tr>
<td>Security Context [50, 540]</td>
<td>📌</td>
<td>🔒</td>
</tr>
<tr>
<td>Security Logger/Auditor [144, 161]</td>
<td>📌</td>
<td>🔒</td>
</tr>
<tr>
<td>Security Policy: A Design Pattern for Mobile Java Code [318]</td>
<td>📌</td>
<td>🔒</td>
</tr>
<tr>
<td>Security Provider [401]</td>
<td>📌</td>
<td>🔒</td>
</tr>
<tr>
<td>Security Session [437]</td>
<td>📌</td>
<td>🔒</td>
</tr>
<tr>
<td>Server Sandbox [282, 540]</td>
<td>📌</td>
<td>🔒</td>
</tr>
<tr>
<td>Session [475, 538, 540]</td>
<td>📌</td>
<td>🔒</td>
</tr>
<tr>
<td>Session Failover [475, 540]</td>
<td>📌</td>
<td>🔒</td>
</tr>
<tr>
<td>Session Scope [475]</td>
<td>📌</td>
<td>🔒</td>
</tr>
<tr>
<td>Session Timeout [475, 540]</td>
<td>📌</td>
<td>🔒</td>
</tr>
</tbody>
</table>

Table C.6 – Software-security patterns classified by security aspects and recognition needs (6).
### Appendix C

**Table C.7** – Software-security patterns classified by security aspects and recognition needs (7).

<table>
<thead>
<tr>
<th>Pattern Name</th>
<th>Structural</th>
<th>Behavioral</th>
<th>Generic Concept</th>
<th>Access Control</th>
<th>Accountability</th>
<th>Availability</th>
<th>Confidentiality</th>
<th>Integrity</th>
<th>Non-Repudiation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Session-Based Attribute-Based Authorization [149]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Session-Based Role-Based Access Control [144, 149]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Share Responsibility For Security [282]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Single Access Point [437, 538, 540]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Single Session [288]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Single Sign-on Delegator [464]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Software Container [473]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Static Integrity Properties [384]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Subject Descriptor [50, 540]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Test on a Staging Server [282]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Third Party Based Authentication [450]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Trusted Proxy [282]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unique Atomic Chunks [201]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unique Entry of Information [208]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Validated Transactions [282]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Virtual Address Space Access Control [142, 144]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Virtual Address Space Structure Selection [144, 154]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Virtual Machine Environment (VME) [474]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Virtual Machine Operating System Architecture [144]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
SECTION DEFINITIONS OF SOFTWARE-SECURITY PATTERNS

In the following all collected sections used in the software-security pattern descriptions are itemized. The square brackets ([]) contain the number of software-security pattern-publications that use this section. The selected definition for the comparison described in Chapter 5 is marked with a star (⋆).

Abstract [1]

⋆ The abstract section describes the intent of a pattern.

Alias [2]

⋆ “The alias section lists other names by which this security pattern might be known.” [436]

Also Known As [6]

⋆ “If the pattern is commonly known by some different names, they can be mentioned here.” [540]
- “Other well-known names for the pattern, if any.” [173]
- “Other names for the pattern, if any are known.” [72, 115, 437]

Alternatives [1]

⋆ “This neutral relationship indicates that two patterns, A and B, while not identical, are functionally equivalent. If A is implemented, then the system will lose nothing by replacing A with B. A may be substituted for B without impacting the overall system behaviour and quality.” [540]

Applicability [6]

⋆ “What are the situations in which the design pattern can be applied? What are examples of poor designs that the pattern can address? How can you recognize these situations?” [173]
- “The applicability describes under what circumstances the pattern can be used. This includes both the scope of the pattern and the development phase in which the pattern is most easily applied.” [540]
- “A general description of the characteristics a program must have for the pattern to be useful in the design or implementation of the program.” [115]

Benefits [5]

* “Not quite as strong as depends, if pattern A benefits from B, then implementing B will add to the value already provided by implementing A. This might be because B enables extra functionality in A, decreases development time, improves the security added to the system by A, etc.” [540]

Class-Diagram [1]

* The class-diagram section describes the static structure of a pattern.

Classification [1]

* Presents the classification of the pattern.

Collaborations [4]

* “How the participants collaborate to carry out their responsibilities.”[173]

Conflicts [1]

* “If pattern A conflicts with pattern B, then implementing B in a system that contains A will result in inconsistencies. It is not useful to implement both patterns to solve the same problem.” [540]

Consequences [40]

* “The consequences section indicates the benefits and liabilities of a solution embodied in this pattern. The benefits should match the forces in the Problem section. Benefits that do not correspond to any force may appear.” [144]
  - “The benefits the pattern provides, and any potential liabilities.” [72, 115, 437]
  - “The advantages and possible disadvantages of applying the pattern.” [540]
  - “Thus it could be helpful to enlist the consequences of the application of a security pattern. The benefits and drawbacks of a security pattern can be discussed.” [436]
  - “Describes the result of using the security pattern as a safeguard and control measure. It also describes the trade-offs.” [464]
  - “They are the effects of the compromise resulting from the application of the pattern’s solution. In particular, cases using security patterns implies an increase in cost (economic, more complex mechanisms, etc.).” [97]

Context [34]

* “We define the context in which the pattern solution is applicable. We may explain relevant characteristics of this context.” [144]
  - “Based on a scenario the context of the security pattern is illustrated. The general conditions under which the problem does occur and which forces do emerge are described. It is useful to list context setting security patterns, too.” [436]
  - “The situations in which the pattern may apply.” [72, 437]
Appendix D

Contraindications [1]
* Describes a situation when the pattern should not be used.

Dependencies [1]
* “This is the strongest reinforcement relationship. If pattern A depends on pattern B, then A will not function correctly without B.” [540]

Discussion [1]
* Discusses the presented solution regarding the presented challenges.

Dynamics [21]
* “Typical scenarios describing the run-time behavior of the pattern.” [72, 437]
- “We next describe (…) some dynamic aspects in the form of sequence diagrams for a use case.” [144]

Example [26]
* “An example of the application of the pattern. This example should provide an easy case to map the upcoming solution to.” [540]
- “In order to illustrate the application of the pattern, concrete examples could be provided. Useful are code or configuration samples as well as some sketches.” [436]
- “A real-world example demonstrating the existence of the problem and the need for the pattern.” [72, 115, 437]
- “We give an example of a problem situation where the use of this pattern may provide a solution.” [144]

Example Instances [1]
* Example instances of the pattern are described in this section.

Example Resolved [19]
* “Discussion of any important aspect for resolving the example that is not yet covered in the Solution, Structure, Dynamics, and Implementation section.” [72, 437]
- “An example of how the real-world example problem described in the Example section may be resolved through the use of the secure design pattern.” [115]
- “Now we can see what happens in the example after a pattern solution has been applied.” [144]

Features [1]
* “They are additional characteristics to the patterns’ properties. They are additional criteria in selecting the suitable patterns.” [97]

Forces [17]
* “Describes the motivations and constraints that affect the security problem. Highlights the reasons for choosing the pattern and provides justification.” [464]
Impairments [1]

* “If pattern A is impaired by B, then the correct functioning of A might be hampered by implementing B. This does not mean that it is impossible to implement both A and B together, but care must be taken that this does not result in errors.” [540]

Implementation [22]

* “The objective of this section is to describe what one should consider when implementing the pattern. This can be a set of general recommendations or a sequence of what to do to use the pattern. It may include some sample code, if appropriate. It is possible to add details of how some products implement this pattern, for example how a particular firewall is implemented by a specific company.” [144]

- “In this section, clues or ideas about the implementation can be given, possibly including sample code. When alternative implementation methods exist for the pattern, these should be mentioned here.” [540]
- “Guidelines for implementing the pattern.” [72, 115, 437]
- “What pitfalls, hints, or techniques should you be aware of when implementing the pattern? Are there language-specific issues?” [173]

Implementation Discussion [1]

* Discusses the presented pattern implementation.

Implementation Issues [1]

* Presents a mixture of design issues and cross-references to standards.

Intent [18]

* “The problem solved by the design pattern and its general rationale and purpose.” [115]

- “The intent of the pattern should be given in a concise way, i.e., what is the purpose of the pattern? What problem does it solve? ” [540]
- “A short statement that answers the following questions: what does the design pattern do? What is its rationale and intent? What particular design issue or problem does it address?” [173]

Issues [1]

* Describes issues that should be considered during the design phase.

Known Uses [37]

* "Examples of the use of the pattern, taken from existing systems." [72, 115, 437]
- “In this part, successful uses of the pattern are given.” [540]
- “Examples of the pattern found in real systems.” [173]
- “To accept this solution as a pattern, we should find at least three examples of its use in real systems. We occasionally break this rule, for example when we see that the solution is clearly generic.” [144]
Appendix D

Labels [1]

* “As a part of our contribution, the labels of the pattern describe the impact (both positive and negative) of the pattern on different qualities such as performance and usability, as well as the impact of the pattern on other security objectives.” [540]

Liabilities [4]

* Liabilities that have to be considered by using the pattern.

Motivation [7]

* “A description of situations in which the pattern may apply and a more detailed description of the problem that the pattern is intended to solve.” [115]
  - “A scenario that illustrates a design problem and how the class and object structures in the pattern solve the problem. The scenario will help to understand the more abstract description of the pattern that follows.” [173]

Name [48]

* “Certainly security patterns aren’t different to “normal” patterns with regard to their name. The name of the pattern becomes a part of the vocabulary of the community. It should be easy to remember and refer to. A good name should be evocative and give an image of what the pattern might be about.” [436]
  - “The name and a short summary of the pattern.” [72, 437]
  - “The pattern name should describe the security pattern in a short but clear, depicting way.” [540]
  - “Provides a memorable and descriptive way to refer to the pattern.” [50]
  - “The pattern’s name conveys the essence of the pattern succinctly.” [173]

Non Security Known Uses [1]

* Examples of non-security pattern usage in existing systems.

Non-Software Example [1]

* An example of using the pattern beyond software implementations.

Participants [6]

* “The class and/or objects participating in the design pattern and their responsibilities.” [173]
  - “The entities involved in the pattern.” [115]

Participants and Responsibilities [1]

* Describes participating objects/classes and their responsibilities (see also Participants).

Pattern Structure [1]

* Presents the static and dynamic structure of the pattern.
Pitfalls [1]

* “The application of the pattern might include some (possibly subtle) pitfalls and risks. Known pitfalls can be mentioned here, optionally including a possible solution.” [540]

Preconditions [1]

* “They indicate assumptions and restrictions related to the deployment of the pattern. Before applying a pattern, users or applications in some cases should check the satisfiability of these pre-conditions. Obviously, pre-conditions are elements used during the selection of suitable patterns for a particular problem.” [97]

Problem [40]

* “The problem for which the pattern offers a solution. As part of this description, the different forces which — by their conflicting nature — lead to the problem can be mentioned as well.” [540]
- “We follow the context with a generic description of what happens when we don’t have a good solution. We also indicate the forces that affect the possible solution.” [144]
- “Describes the security issues addressed by the pattern.” [464]
- “The problem the pattern addresses, including a discussion of its associated forces.” [72, 437]
- “A description of the contexts and situations in which the pattern is useful.” [50]
- “The problem statement defines the problem that will be solved by the security pattern. The major aspects of the problem are elaborated by the viewpoint of the forces to be solved. In the field of security, a problem occurs whenever a system component is protected in an insufficient way against abuse.” [436]

Problem/Requirements and Context [1]

* “The problem is the vulnerable part in an asset that can also be described as requirements which need to be solved. The context defines the recurring situation where they can occur.” [97]

Properties [1]

* “They describe which security elements the pattern is providing. This is the basic element used to discriminate whether a pattern is useful for a security problem or not.” [97]

Rationale [1]

* Describes how the pattern resolves the mentioned forces.

Reality Checks [1]

* “Describes a set of review items to identify the feasibility and practicability of the pattern.” [464]

Related Patterns [33]

* “Lists other related patterns from the security pattern catalog or from other related sources.” [464]
- “As some countermeasures may introduce other vulnerabilities, additional security patterns should be considered in the related patterns section, too. The same is valid for problems that are solved partly or couldn’t be considered within the given security pattern. That way a pattern hierarchy will be formed.” [436]
- “The relations of the pattern with other patterns are noted down here.” [540]
- “What design patterns are closely related to this one? What are the important differences? With which other patterns should this one be used?” [173]

Relationships [1]
* “A short summary of the inter-pattern relationships should be given.” [540]

Resultant Context [1]
* Describes the environment that is created by applying the solution (see also Resulting Context).

Resulting Context [1]
* Describes the environment that is created by applying the solution (see also Resultant Context).

Sample Code [5]
* “Code fragments that illustrate how you might implement the pattern.” [173]
- “Code providing an example of how to implement the pattern.” [115]

Security Factors and Risks [1]
* “Describes the factors and risks to be considered while applying the pattern.” [464]

Security Objectives [1]
* “The objectives describe the main security objective the pattern tries to solve. Multiple objectives can be given, but this should be rare. If a pattern has an influence on another than its main objective, this can be mentioned in the labels section.” [540]

See Also [8]
* “Finally, we relate our pattern to other known patterns. Those may be complementary patterns, variations of our pattern or extensions of it.” [144]
- “References to patterns that solve similar problems, and to patterns that help us refine the pattern we are describing.” [72, 437]

Social Dependencies [1]
* “An agent most likely participates in relationships with other agents. In many traditional software engineering techniques, relationships are focused only on the exchange of data and intended functions. However, agent relationships are similar to human relationships and thus much more complex. Agent relationships involve conflicts amongst the relationships, multi-lateral relationships, and delegation of relationships.” [342]
Solution [41]

- "The fundamental solution principle underlying the pattern." [72]
- "The solution section describes the idea of the pattern. A descriptive figure may help to visualize the solution." [144]
- "The solution is defined as a mechanism that is used to resolve the corresponding requirement/problem. It defines the sequential flow of operations in solving the security problem." [97]
- "This section describes the solution of the problem. Appropriate solutions are determined by the context of the pattern. According to certain security objectives (that may be written down in security policies), countermeasures have to be applied in order to reduce the risk. It is useful to warn from pitfalls (how does this pattern become an anti-pattern) and refer to variants of the pattern." [436]
- "A specific but flexible approach to solving the problem." [50]
- "A complete and detailed description of the solution provided by the pattern. This description is comprised of the static structure and the dynamic behavior of the solution (preferably in a graphical notation like UML), the participants together with their responsibilities and, finally, the collaborations between the participants." [540]
- "Describes the approach briefly and the associated mechanisms in detail." [464]

Strategies [1]

- "Describes different ways a security pattern may be implemented and deployed." [464]

Structure [27]

- "We next describe the structure (static view) of the solution (...) for a use case." [144]
- "A detailed specification of the structural aspects of the pattern, including CRC-cards for each participating component and an OMT class diagram." [72]
- "A detailed specification of the structural aspects of the pattern, using appropriate notations." [437]
- "Describes the basic structure of the solution using UML sequence diagrams and details the participants." [464]
- "A textual or graphical description of the relationship between the various participants in the pattern. This provides a detailed specification of the structural aspects of the pattern, using appropriate notations." [115]
- "A graphical representation of the classes in the pattern using a notation based on the Object Modeling Technique (OMT). We also use interaction diagrams to illustrate sequences of requests and collaborations between objects." [173]

Structure and Participants [1]

- Depicts the structure of the solution including the participants.

Summary [1]

- Summarizes briefly the problem and context of a pattern (see also Thumbnail).

Thumbnail [1]

- Describes briefly the problem and context of a pattern (see also Summary).
Appendix D

Trade-Offs [1]

★ Describes the contributions of a pattern on the following security goals: accountability, availability, confidentiality, integrity, manageability, usability, performance, and cost.

Variants [6]

★ “A brief description of variants or specializations of a pattern.” [72, 437]

Vulnerability Analysis [1]

★ Presents a vulnerability analysis of an unsecured pattern and compares it to a secured pattern version.
RESULT OF THE MOBILE SECURITY GUIDELINE MAPPING

The tables E.1 and E.2 present the collected security patterns mentioned in the mobile security guidelines of Section 8.2.3. The x-mark icon (✗) highlights that a security pattern could be mapped to a described security issue within the guideline.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Information Obscurity [437]</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>5</td>
</tr>
<tr>
<td>Secure Channel [437]</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>5</td>
</tr>
<tr>
<td>Secure Pipe [464]</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>5</td>
</tr>
<tr>
<td>Secure Communication [50]</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>5</td>
</tr>
<tr>
<td>Authentication Enforcer [464]</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>5</td>
</tr>
<tr>
<td>Credentials [341]</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>4</td>
</tr>
<tr>
<td>Credential Tokenizer [464]</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>3</td>
</tr>
<tr>
<td>ID/Password Authentication [357]</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>3</td>
</tr>
<tr>
<td>Password Authentication [282]</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>3</td>
</tr>
<tr>
<td>Path Canonicalization [115]</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>3</td>
</tr>
</tbody>
</table>

**Table E.1** – Collected guideline information mapped to security patterns (1).
### Security Pattern

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Clear Sensitive Information [115]</td>
<td>✗</td>
<td>✗</td>
<td></td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>Cryptographic Metapattern [59]</td>
<td>✗</td>
<td>✗</td>
<td></td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>File Authorization [151]</td>
<td>✗</td>
<td>✗</td>
<td></td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>Input Validation [115]</td>
<td>✗</td>
<td>✗</td>
<td></td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>Remote Authenticator [153]</td>
<td>✗</td>
<td>✗</td>
<td></td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>Sandbox [342]</td>
<td>✗</td>
<td>✗</td>
<td></td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>Access Controller [342]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Authenticated Session [115]</td>
<td>✗</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Certificate Authority [303]</td>
<td></td>
<td></td>
<td>✗</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Certificate Revocation [303]</td>
<td></td>
<td></td>
<td>✗</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Client Input Filters [115]</td>
<td></td>
<td></td>
<td></td>
<td>✗</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Cryptographic Smart Card [303]</td>
<td>✗</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Input Guard [424]</td>
<td></td>
<td></td>
<td></td>
<td>✗</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Key in the Pocket [302]</td>
<td></td>
<td></td>
<td></td>
<td>✗</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Message Integrity [59]</td>
<td></td>
<td></td>
<td></td>
<td>✗</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Password Synchronizer [464]</td>
<td>✗</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Policy [538]</td>
<td></td>
<td></td>
<td></td>
<td>✗</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Policy Based Access Control [106]</td>
<td></td>
<td></td>
<td></td>
<td>✗</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Secure Logger [464]</td>
<td>✗</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Security Logger / Auditor [161]</td>
<td></td>
<td></td>
<td></td>
<td>✗</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Security Policy [318]</td>
<td></td>
<td></td>
<td></td>
<td>✗</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Single Sign On Delegator [464]</td>
<td></td>
<td></td>
<td></td>
<td>✗</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Sender Authentication [59]</td>
<td></td>
<td></td>
<td></td>
<td>✗</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Session [538]</td>
<td>✗</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Session Timeout [475]</td>
<td></td>
<td></td>
<td></td>
<td>✗</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Symmetric Encryption [225]</td>
<td>✗</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

Table E.2 – Collected guideline information mapped to security patterns (2).
The following points were discussed during the telephone conference with SAFECode members [414]:

- Based on your experience: Are these steps essential to assess security features? Is there anything missing? (With regard to Figure 1.2)
- Usage: Can you imagine that this tool can enhance the work of analysts?
- Information presentation: What can make the usage of such a graph easier? Color highlighting? Search function? Hiding unwanted node types? Clustering?
- Tool evaluation: Which participant skills are required? Is it sufficient to evaluate with students to obtain representative results?
- Can static tool usage replace a manual review or at least support it?