UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft

Faculty for Computer Science, Electrical Engineering and Mathematics

CogniCrypt – The Secure Integration of Cryptographic Software

Stefan Krüger

Doctoral Thesis

Submitted in partial fulfillment of the requirements for the degree of

Doktor der Naturwissenschaften (Dr. rer. nat.)

Advisors
Prof. Dr. Eric Bodden
Prof. Dr. Karim Ali

Paderborn, October 15, 2020

Abstract

Prior research has empirically laid bare the widespread misuse of cryptographic APIs in software applications. Developers struggle with bad API design and lack of cryptographic knowledge when they attempt to implement cryptographic features into their applications. Several approaches and tools have been proposed to solve this issue, but all fall short in addressing developers' needs in one way or another. This results in a landscape of patchy solutions.

In this thesis, we address the issue of cryptographic misuse more systematically through CogniCrypt. CogniCrypt integrates different kinds of tool support into a unified approach in order to lift the burden of needing to know how to use cryptographic APIs from the developer. Front and center in our approach is CrySL, a specification language for bridging the cognitive gap between cryptography experts and developers. CrySL enables cryptography experts to specify the secure usage of the cryptographic APIs they develop. We have implemented a compiler for CrySL that facilitates building API tool support on top of CrySL specifications. We have further devised extensive CrySL rule sets for several cryptographic APIs in Java.

As a first type of tool support, we present the context-sensitive and flow-sensitive demand-driven static analysis CogniCrypt$_{\text{SAST}}$. CogniCrypt$_{\text{SAST}}$ helps developers by automatically checking a given application for compliance with the CrySL-encoded rules. We have empirically evaluated CogniCrypt$_{\text{SAST}}$ by analyzing 10,000 Android apps and 204,788 current Java software artefacts on Maven Central. Our findings confirm previous results with 95% of apps and 63% of Maven artefacts containing at least one misuse.

We have further devised CogniCrypt$_{\text{GEN}}$, a code generator that proactively assists developers in using Java cryptographic APIs correctly. Prior attempts to address misuses focused on detecting them after the fact. CogniCrypt$_{\text{GEN}}$, on the other hand, supports developers avoiding such misuses in the first place and may therefore help significantly reducing development cost. CogniCrypt$_{\text{GEN}}$ accepts as input a code template and CrySL rules. The code templates in CogniCrypt$_{\text{GEN}}$ are minimal, only comprising simple glue code. All security-sensitive code is generated fully automatically from CrySL rules that the templates merely refer to. That way, generated code is provably correct and secure with respect to CrySL definitions. CogniCrypt$_{\text{GEN}}$ supports the implementation of the most common cryptographic use cases, ranging from password-based encryption to digital signatures. We have empirically evaluated CogniCrypt$_{\text{GEN}}$ from the perspectives of both cryptographic-API and application developers. Our results show that CogniCrypt$_{\text{GEN}}$ is fast enough to be used during development. Furthermore, it requires minimal development and maintenance effort and no further programming-language skills beyond Java. Real-world developers see CogniCrypt$_{\text{GEN}}$ as significantly simpler to use than a comparative template-based solution.

We have implemented CogniCrypt as an Eclipse plugin that combines CogniCrypt$_{\text{SAST}}$ and CogniCrypt$_{\text{GEN}}$ and conducted a controlled experiment to evaluate the tool's effectiveness. The study included 24 participants who we asked to implement to short programming tasks,

one with CogniCrypt, and one with a regular Eclipse without CogniCrypt. We measured the functional correctness and security of participants' code as well as their completion rate and time. We also asked participants for their feedback on CogniCrypt's usability. Our findings show that study participants fared significantly better in all respects with CogniCrypt than without. Participants further significantly preferred CogniCrypt over regular Eclipse. These results provide strong evidence that CogniCrypt does accomplish the aspired goal of this thesis: Effectively combatting cryptographic misuse by relieving software developers of having to know how to use cryptographic APIs.

Zusammenfassung

Frühere Studien haben empirisch offenbart, dass Fehlbenutzungen von kryptographischen APIs in Softwareanwendungen weitverbreitet sind. Dies geschieht vor allem, weil Software-Entwickler_innen aufgrund schlechten API-Designs und fehlenden Kryptographiewissens Probleme bekommen, wenn sie versuchen kryptographische Features zu implementieren. Die Literatur liefert mehrere Ansätze und Vorschläge diese Probleme zu lösen, aber alle scheitern schlussendlich auf die eine oder andere Weise daran die Anforderungen der Entwickler_innen zu erfüllen. Das Resultat ist eine insgesamt lückenhafte Landschaft verschiedener nur wenig komplementärer Ansätze.

In dieser Arbeit adressieren wir das Problem kryptographischer Fehlbenutzungen systematischer durch CogniCrypt. CogniCrypt integriert verschiedene Arten von Tool Support in einen gemeinsamen Ansatz, der Entwickler_innen davon befreit wissen zu müssen, wie diese APIs benutzt werden müssen. Zentral für unseren Ansatz ist CrySL, eine Beschreibungssprache, die die kognitive Lücke zwischen Kryptographie-Expert_innen und Software-Entwickler_innen überbrückt. CrySL ermöglicht es Kryptographie-Expert_innen zu spezifizeren, wie die APIs, die sie bereitstellen, richtig benutzt werden. Wir haben einen Compiler für CrySL implementiert, der es erlaubt auf CrySL-Spezifikationen aufbauenden Tool Support zu entwickeln. Wir haben weiterhin umfassende CrySL-Regelsätze für mehrere Crypto-APIs in Java erstellt.

Als ersten CrySL-basierten Tool Support präsentieren wir die statische, kontext- und flusssensitive On-demand-Programmanalyse CogniCrypt$_{\text{SAST}}$ als Teil dieser Arbeit. CogniCrypt$_{\text{SAST}}$ unterstützt Entwickler_innen, indem es automatisch eine Anwendung darauf überprüft, ob sie sich an CrySL-Regeln hält. Wir haben CogniCrypt$_{\text{SAST}}$ durch eine Analyse von 10.000 Android-Apps und 204.788 Java-Programmen auf Maven Central evaluiert. Unsere Ergebnisse sind im Einklang mit früheren Studien: 95% der Android-Apps und 63% der Maven-Anwendungen beinhalten mindestens eine Fehlbenutzung.

Wir haben außerdem CogniCrypt$_{\text{GEN}}$ entworfen. CogniCrypt$_{\text{GEN}}$ ist ein Code Generator, der Entwickler_innen proaktiv dabei hilft kryptographische APIs zu nutzen. Frühere Ansätze haben sich vor allem darauf fokussiert, Fehlbenutzungen zu finden. CogniCrypt$_{\text{GEN}}$ ermöglicht es Entwickler_innen solche Fehlbenutzungen komplett zu vermeiden und kann dadurch Entwicklungskosten signifikant reduzieren. CogniCrypt$_{\text{GEN}}$ verarbeitet ein Java-Code-Template und CrySL-Regeln. Die Code-Templates in CogniCrypt$_{\text{GEN}}$ sind minimal, da sie lediglich einfach Glue Code beinhaltet. CogniCrypt$_{\text{GEN}}$ generiert allen sicherheitsrelevanten Code automatisch aus CrySL-Regeln. Dadurch ist der Code beweisbar korrekt und sicher in Bezug zu CrySL-Definitionen. CogniCrypt$_{\text{GEN}}$ unterstützt die Implementierung der häufigsten kryptographischen Anwendungsfälle, von passwort-basierter Verschlüsselung bishin zu digitalen Signaturen. Wir haben CogniCrypt$_{\text{GEN}}$ empirisch sowohl aus Perspektive von Kryptographie-Expert_innen als auch jener von Software-Entwickler_innen evaluiert. Unsere Ergebnisse zeigen, dass CogniCrypt$_{\text{GEN}}$ schnell genug ist um während alltäglicher Entwicklungsarbeit eingesetzt zu wer-

den. Es erfodert auch nur minimalen Entwicklungs- und Wartungsaufwand und keine weiteren über Java hinausgehenden Programmiersprachenkenntnisse. Entwickler_innen schätzen CogniCrypt$_{GEN}$ als signifikant nützlicher ein denn eine ähnliche ebenfalls template-basierte Lösung.

Wir haben CogniCrypt schlussendlich als Eclipse-Plugin implementiert, das CogniCrypt$_{GEN}$ und CogniCrypt$_{SAST}$ kombiniert. Um die Effektivität des Tools zu evaluieren, haben wir ein kontrolliertes Experiment mit dem Tool durchgeführt. An dem Experiment nahmen 24 Teilnehmer_innen teil, denen wir auftrugen zwei Programmieraufgaben zu absolvieren, eine mit CogniCrypt, eine mit einem regulären Eclipse. Während der Studie maßen wir sowohl wie viele Teilnehmer_innen die Aufgaben erfolgreich implementierten und wie lange sie dafür brauchten als auch die funktionale Korrektheit und Sicherheit der Lösungen der Teilnehmenden. Wir baten die Teilnehmer_innen auch uns Feedback bezüglich CogniCrypt s Nutzbarkeit zu geben. Die Ergebnisse des Experiments zeigen, dass Teilnehmende in allen Belangen besser abschnitten, wenn sie CogniCrypt nutzten. Die Teilnehmer_innen bevorzugten CogniCrypt auch signifikant gegenüber dem regulären Eclipse. Diese Ergebnisse liefern starke Beweise, dass CogniCrypt es schafft, die Ziele, die wir uns innerhalb dieser Arbeit gesetzt haben zu erfüllen: Effektiv die Fehlbenutzung kryptographischer APIs zu bekämpfen, indem es Software-Entwickler_innen die Bürde abnimmt, wissen zu müssen, wie kryptographische APIs sicher benutzt werden.

Acknowledgements

I would first like to express gratitude to my two advisors Eric Bodden and Karim Ali. Without them and their support throughout my Ph.D., this thesis would not have been possible. I have greatly benefited from both their research expertise as well as their striving to provide a constructive and safe work environment. Eric's approachability has always been a source of trust in him for me. Similarly, especially in the beginning, Karim managed to calmly give my work the structure that it had previously lacked.

I also want to thank my colleagues from Paderborn University and TU Darmstadt. In particular, I am grateful to Johannes Späth. Without his work on and contributions to this research, it would not be in nearly as good a shape as it is now. I also very much enjoyed and sometimes still think back to our seemingly endless discussions when we worked together. I would further like to thank Lisa Nguyen Quang Do for having been there when it counted. I have also always appreciated her unbreakable desire to get things right. I am also grateful to Ben Hermann who I have always enjoyed conversing with and who I have great respect for. I still hope, we will get the chance to write a follow-up paper to our work for GE. I am also thankful to Sarah Nadi who I very much enjoyed working with in the beginning of my Ph.D. To this day, I am impressed by her structured style of working and her writing. Thanks also to Martin Mory, Philipp Holzinger, Philipp Schubert, Michael Reif, Anna-Katharina Wickert, Johannes Geismann, Vera Meyer, and Sammy Maniera for their companionship over the past few years. Special thanks to those among my colleagues who very generously allowed me to use their showers when mine was broken.

Thanks further to Eric Bodden, Karim Ali, Mira Mezini, Sarah Nadi, Johannes Späth, Michael Reif, Daniel Demmler, Felix Günther, Christian Weinert, and Florian Göpfert as well as the members of project group SICS for their contributions to the research that eventually culminated in this thesis. I would also like to thank all students who have contributed to the implementation work around CogniCrypt over the years: André Sonntag, Patrick Hill, Sneha Reddy, Shahrzad Asgharivaskasi, Enri Ozuni, Seena Mathew, Rakshit Krishnappa Ravi, Sriteja Kummita, Gökçe Karakaya, Anemone Kampkötter, Taslima Akter, Adnan Manzoor, and Mohammad Zahree. Lastly, thanks also to those who gave insightful and valuable feedback to my journal, conference, and workshop papers and all participants of my two user studies.


CogniCrypt

Contents

1 Introduction

Digital devices are widely used for storing sensitive data. Cryptography is the primary means to protect such data from eavesdropping or forgery. For this protection to be effective, the used cryptographic algorithms must first be conceptually secure, that is, there must exist no attacks to break them. Most algorithms get broken eventually through thorough cryptanalysis or thanks to advancing computing power, but a wide range of still-secure algorithms is available at any given time. Moreover, cryptographic algorithms must be implemented securely in the used programming language. A security-breaking bug in an implementation makes all software that relies on this implementation vulnerable to the same attack. Patches may fix the vulnerability, but any version up to the patch may no longer be used and its use must be phased out quickly. Rolling out such patches is no easy endeavour, even less so when cryptographic implementations are shipped as part of a programming language's development kit.

While these problems are indeed still of relevance to contemporary cryptography, another problem has arisen and it appears to impact the security of actual cryptographic applications to an even greater extent. Lazar et al. [LCWZ14] investigated 269 cryptography-related vulnerabilities and found that only 17% are related to insecure implementations of algorithms. In stark contrast, 223 of these vulnerabilities (83%) are caused by application developers failing to integrate cryptographic algorithms into their code. All research investigating this issue concludes that the phenomenon is vast in size [EBFK13, KSA+19b, CNKX16, SDG+14, RXA+19, MBB18, GWL+19, WLZ+17, LZLG14, Fei19, ZCD+19, Ver17]. The specific reasons behind individual misuses may be manifold, but the design of cryptographic libraries and their application programming interfaces (API) appear to take a central role in why the misuse is as widespread as it is. While some of the APIs have been in active development for a decade or more, their use is largely still far from straightforward for developers as the following example illustrates.

1.1 A Motivating Example

```
1  public Key generateKey(String pwd) {
2    byte[] salt = {15, -12, 94, 0, 12, 3, -65, 73, -1, -84, -35};
3    PBEKeySpec spec = new PBEKeySpec(pwd.toCharArray(), salt, 100000, 256);
4
5    SecretKeyFactory skf =
         SecretKeyFactory.getInstance("PBKDF2WithHmacSHA256");
6    byte[] keyMaterial = skf.generateSecret(spec).getEncoded();
7
8    SecretKeySpec cipherKey = new SecretKeySpec(keyMaterial, "AES");
9    return cipherKey;
10 }
11
12 public byte[] encrypt(byte[] plaintext, Key cipherKey) {
13     Cipher ciph = Cipher.getInstance("AES");
14     ciph.init(Cipher.ENCRYPT_MODE, cipherKey);
15     return ciph.doFinal(plaintext);
16 }
17
18 public byte[] decrypt(byte[] ciphertext, Key cipherKey) {
19     Cipher ciph = Cipher.getInstance("AES");
20     ciph.init(Cipher.DECRYPT_MODE, cipherKey);
21     return ciph.doFinal(ciphertext);
22 }
```

Figure 1.1: An Example Illustrating an *Incorrect* Implementation of a Password-based Encryption (PBE) in Java.

For the purpose of password-based data encryption (PBE), the Java Cryptography Architecture (JCA) [Inc17], the most widely used cryptographic library for Java [NKMB16], offers APIs `PBEKeySpec` and `Cipher`. The former is used for the derivation of a cryptographic key from a password, the latter for the actual encryption. Figure 1.1 presents a potential use of these classes that is can be found in numerous real-world applications.

The method `generateKey()` begins with the setup of the encryption key by creating a `PBEKeySpec` object. The constructor of `PBEKeySpec` expects a password, a salt, an iteration count, and a key length. The `PBEKeySpec` object, once created, is passed to a `SecretKeyFactory` that uses the password-based key-derivation algorithm `PBKDF2` to generate key material (Lines 5–6). Based on this key material, Line 8 generates a key for the symmetric cipher AES. The method `encrypt()` then executes the actual encryption using the class `Cipher`. Line 13 selects the encryption algorithm `AES` to be used. The call to `init()` at Line 14 puts the `Cipher` object in encryption mode and makes it use the previously generated key for the encryption. The last statement of the method completes the encryption by a call to `doFinal()`. Method `decrypt()` reverses this operation to receive the plaintext.

While the example avoids some common misuses (e.g., using a low iteration count [EBFK13, KSA+18, BDA+17, SDG+14, CNKX16]), it contains several less trivial misuses that make the code insecure, three of which are related to `PBEKeySpec`, two more to `Cipher`. Three of the four arguments to the constructor (Line 3) have usage constraints, and the example breaks two of them. Let us first investigate the parameter that it sets correctly: the iteration count. There is no general consensus on what a secure iteration count is, but 100,000 is well above most recommendations [GGF17, fISB17, Nat18]. The first misuse relates to the constant salt that is passed to the constructor call. Salts should be generated through a cryptographically

secure random source to keep them unpredictable [GGF17, fISB17]. The second misuse is for the argument `pwd`. The constructor of `PBEKeySpec` expects the password to be of type `char[]`, and it does so for a good reason. Passwords should not remain accessible in memory any longer than absolutely necessary. Unlike arrays, strings are immutable in Java. Whenever a string is modified, a new string is created, and the old one is kept in memory until garbage-collected. To limit the password's lifetime in memory, developers must favour using `char[]` over `String` for passwords and clear the array after passing it to the constructor [Ora19b]. The third misuse is related to the fact that `PBEKeySpec` does not automatically clear the password soon after use. Instead, it expects the developer to call `clearPassword()` after the object has fulfilled its purpose, which the code in Figure 1.1 does not do.

Method `encrypt()` exhibits a common misuse [EBFK13, KSA+19b]. The method `Cipher.getInstance()` does not just expect an algorithm, but also a mode of operation and padding scheme. Block ciphers, such as AES, the algorithm used in the example, require such a mode and padding scheme as they divide plaintexts into blocks and perform the encryption block-by block. To get the full ciphertext, such algorithms subsequently re-assemble the ciphertext blocks and mode of operation and padding scheme configure how this process is handled. If the developer fails to properly specify these two parameters, like in Line 13, the underlying library selects them automatically based on the passed algorithm. Unfortunately, the default JCA provider selects Electronic Code Book mode as mode of operation. This mode is widely considered insecure [fISB17, GGF17] for plaintexts longer than a single block. Fixing just the mode will not suffice, though. All other modes of operation require additional outside data (e.g., initialization vectors, nonces) for secure re-assembling. The developer may either provide this data themselves to the `init()` call in Line 14, or if not, the underlying provider again generates it for them. In the latter case, they must then retrieve it through a call to `Cipher.getIV()` on `ciph`. In either case, they must append this data to the ciphertext to ensure `decryption` is possible.

Although the code contains these security-breaking misuses, it nonetheless runs without throwing exceptions. Not only must developers make sure to use the API in a functionally correct way, they also must consider the code's security properties. This scenario is especially concerning considering that Java does not provide any tool support to detect insecure uses.

1.2 A Broader Perspective

While the issues the example illustrates may look isolated or cherry-picked at first, they are really not. As argued above, crypto(graphic) APIs are frequently misused. A huge body of research on different programming languages backs up this conclusion [EBFK13, KSA+19b, CNKX16, SDG+14, RXA+19, MBB18, GWL+19, WLZ+17, LZLG14, Fei19, ZCD+19, Ver17]. As a result, despite the availability of mature, (still-)secure-to-use, and securely-implemented cryptographic algorithms, the vast majority of cryptographic applications is insecure. These findings highlight the importance of addressing the source of cryptographic insecurity.

As a first step towards this goal, Nadi et al. [NKMB16] previously triangulated the results of four empirical studies to investigate the reasons for the misuse. Two of them survey Java developers familiar with the Java Crypto APIs. The majority of participants (65%) found their respective Crypto APIs hard to use. When asked why, participants mentioned the API level of abstraction (35%), insufficient documentation without examples (43%), and an API design (33%) that makes it difficult to understand how to properly use the API. These results indicate that participants not only lack the domain knowledge, but also struggle with APIs themselves and how to use them. When asked what would help them use these APIs, they suggested better documentation, different API designs, and additional tool support. In terms of API

design, participants used terms such as *use cases*, *task-based*, and *high-level*. These suggestions indicate that developers struggle with the fact that cryptographic APIs reside on the low level of a security primitive API [IG17], i.e., they expose cryptographic algorithms directly to the developer, instead of providing higher-level convenience methods such as `encryptFile()`. When it comes to tool support, participants suggested tools such as *CryptoDebugger*, *analysis tools* that find misuses and provide *code templates* or *generate code* for common functionality.

The literature offers several approaches to fixing cryptographic misuse that roughly follow the participants' suggestions. One particular line of research attempts to improve existing resources such as nudging people towards secure examples on popular online Q&A forums [FXK+19] or designing API documentation to be more example-driven [MW18, HZH19]. These attempts would certainly improve the situation, do have the drawback, however, of having to be adopted by the official vendor of the forum or API, respectively. Others have attempted the more long-term solution of redesigning APIs such that they provide easy-to-use interfaces [BPF19, BLS12, FLW12, Lou, Pytb, Soda, Sodb, Mai, Pyta, Goo, Smi, PTSA+, Cos] or suggesting what such redesigns should look like [GIW+, ABF+17, MKW18, IG17, GS16, PHR19, WvO08, vdLRWW18, KLCL18, IKND16]. These endeavours are worth their time as the mistakes made in the design of many widely used Crypto APIs must not be made again. Discussions on better designs drive the understanding forward and attempts at re-designing allow for usability testing to determine if the suggestions actually work [ABF+17, MKW18]. Yet, the badly-designed APIs do already exist and they are widely used [NKMB16]. Developers are stuck using them for the time being and cannot force a re-design on their own.

Yet other work has attempted to detect misuses of Crypto APIs through program analysis [EBFK13, CNKX16, SDG+14, RXA+19, MBB18, GWL+19, WLZ+17, LZLG14, Fei19, ZCD+19] and program-repair [MLLD16, SZSS19]. While this step offers relief for developers who use existing APIs, existing approaches are insufficient for several reasons. First, these approaches implement mostly lightweight *syntactic checks*, which yield fast analysis times at the cost of missing a high number of false negatives. Therefore, such analyses fail to warn about many insecure uses of cryptography, especially non-trivial ones like the ones we discussed above for Figure 1.1. Moreover, existing tools cannot easily be extended to cover those rules; instead they have *hard coded* cryptography-specific usage rules. The JCA offers a plugin design that enables different providers to offer different crypto implementations through the same API, often imposing slightly different usage requirements on their clients. Hard-coded rules can hardly reflect this diversity. Such detection and mitigation techniques also come at a cost when used on their own: developers first must integrate the API insecurely to then—hopefully, at some point—learn how to fix the integration, a request that seems questionable when taking into account the results of the study by Nadi et al. [NKMB16] and others [EBFK13, GKL+19, PTRV18, TJVW19, NDT+17, NDTS18, NDG+19]. We conclude that existing work on fixing crypto-API misuse is quite patchy. Several different and independent approaches in different directions may provide short-term relief for some cryptographic misuse or a promise for a better future, but there is no comprehensive direct support.

This thesis takes a different approach at addressing cryptographic misuse. We recognize that application developers require more immediate help. As a result, we design our approach to help developers while they are working with these APIs right now. From the previous discussion, we also conclude that developer support should come from different angles. Just a program analysis or just better documentation do not suffice. Instead, our solution will integrate multiple types of support into a single unified approach. In summary, our thesis follows this goal:

Crypto-API misuse can be effectively combatted through an integrated approach that relieves the software developer of having to know how to use the API.

1.3 Contributions of the Thesis

In this thesis, we present CogniCrypt, an integrated approach to addressing misuse of Crypto APIs. To smoothly integrate into developers' workflows, we have designed and prototypically implemented CogniCrypt as a plugin for the IDE Eclipse. For the purpose of this thesis, we divide CogniCrypt into three individual contributions. At the heart of our approach lies CrySL, this thesis' first contribution. CrySL is a textual specification language that enables domain experts to formally specify how to use an API. By means of CrySL, such experts may express in which order methods are to be called, mark methods as deprecated and otherwise forbidden, prohibit no-longer or never-secure algorithms, as well as define requirements for how two classes may (not) be composed. We have modeled the most commonly used cryptographic APIs in Java using CrySL. On top of CrySL, different kinds of tool support that aid developers in using the covered APIs may be built, two of which we further present in this thesis.

The first such tool support is CogniCrypt$_{\text{SAST}}$, a static program analyzer that checks a given Java or Android app for compliance with the encoded CrySL rules. To this end, it translates rules in CrySL into an efficient, yet precise flow-sensitive and context-sensitive static data-flow analyses. CogniCrypt$_{\text{SAST}}$ goes beyond methods that are useful for general validation of API usage (e.g., typestate analysis [AAC$^+$05, BA07, NL08, Bod10] and data-flow checks [ABB$^+$09, ARF$^+$14]) by enabling the check of domain-specific constraints related to cryptographic algorithms and their parameters. We have evaluated CogniCrypt$_{\text{SAST}}$ in three studies. First, we have run it on 10,000 randomly selected Android apps. Our study reveals 95% of Android apps to contain at least one misuse. Second, we have applied the tool to 250 Android apps from security-critical domains, of which we find 71% to misuse Crypto APIs. In our third study, we have executed CogniCrypt$_{\text{SAST}}$ on 204,788 artefacts on Maven Central. Across all analyzed artefacts that use cryptographic APIs, CogniCrypt$_{\text{SAST}}$ finds 24,349 cryptography misuses in 5,712 Java artefacts. More than 63% of all artefacts that use the JCA contain at least one misuse.

This thesis further contributes CogniCrypt$_{\text{GEN}}$, a code generator for secure integrations of Crypto APIs. The approach operates on a Java project into which it generates code, and accepts as input a template with interface and glue code, as well as usage rules in CrySL. Our design of CogniCrypt$_{\text{GEN}}$ simplifies the used code templates by having the vast majority of the code be generated fully automatically from CrySL. This code is free of syntax errors, type-safe, and provably correct and secure by construction (assuming a correct and secure specification of CrySL rules by the domain experts). We have evaluated CogniCrypt$_{\text{GEN}}$ in terms of expressiveness, maintainability, and usability. This evaluation shows that CogniCrypt$_{\text{GEN}}$ supports the most common cryptographic use cases, with significant lower effort for template developers than in other comparative code generators.

In our final contribution, we evaluate the effectiveness of CogniCrypt. To this end, we conduct a user study with 24 participants using the Eclipse prototype of CogniCrypt. Participants were asked to perform two programming tasks, one with CogniCrypt, one with a regular Eclipse. At the end, they provided feedback about their experience. Our results show that, with CogniCrypt, participants produce not only more secure code but are also significantly faster at producing functional code. In the post-study survey, participants also indicate a strong preference for CogniCrypt over plain Eclipse.

1.4 Structure of the thesis

The remainder of this thesis is structured as follows: In Chapter 2, we give an introduction to concepts necessary to understand the remainder of this thesis, including cryptography and different forms of static data-flow analysis. Chapter 3 presents a high-level overview of CogniCrypt. In Chapter 4, we discuss CogniCrypt's related work. Chapter 5 provides an in-depth discussion of CrySL. To this end, it presents the reasoning behind design decisions made for CrySL, its syntax, and formal semantics. Chapter 6 discusses CogniCrypt$_{\text{SAST}}$. In that chapter, we describe how CogniCrypt$_{\text{SAST}}$ implements CrySL's formal semantics and our two large-scale evaluations. In Chapter 7, we present CogniCrypt$_{\text{GEN}}$. The chapter first describes how CogniCrypt$_{\text{GEN}}$ transforms CrySL specifications into Java code. Later on, it discusses our multi-faceted evaluation of CogniCrypt$_{\text{GEN}}$. Chapter 8 provides a description of a user study evaluating CogniCrypt in the form of its prototypical implementation as an Eclipse plugin. In Chapter 9, we discuss further applications of CrySL to illustrate its expressive power. The thesis concludes with Chapter 10.

The work presented in Chapter 3 has, in parts, been published at the tool track of the 32nd ACM/IEEE international conference Automated Software Engineering (ASE) [KNR+17]. For the purpose of this thesis, we have updated the text from that paper with development that has taken place since publication.

The work in Chapters 5 and 6 has originally been published at the main research track of the 32nd European Conference on Object-oriented Programming (ECOOP) [KSA+18]. This publication does neither include Section 5.4 nor the studies discussed in Sections 6.4 and 6.5. An extended version that does include the latter study has been published in the IEEE Transactions on Software Engineering journal (TSE) [KSA+19b] in 2020. In extension to both these papers, this thesis provides a more in-depth discussion of CrySL's limitations, CogniCrypt$_{\text{SAST}}$'s workflow, and a study of 250 security-critical Android apps that we conducted with CogniCrypt$_{\text{SAST}}$.

Chapter 7 has been published at the 2020 Symposium on Code Generation and Optmisation (CGO) [KAB20].

An extended version of the work presented in Chapter 8 is currently work in progress [KNR+21].

2 Background

In this chapter, we briefly introduce the fundamentals of cryptography, how this cryptography is implemented in Java, and discuss topics on static data flow analysis that are of relevance of to this thesis.

2.1 Cryptography

For the purpose of this thesis, we provide a brief introduction into a set of cryptographic operations, this thesis will discuss. We also discuss the foundations of the transport layer security protocol (TLS). Finally, we introduce how these cryptographic concepts are implemented in Java.

2.1.1 Low-level Cryptographic Operations

Through cryptography, a number of security properties may be ensured: confidentiality, integrity, authenticity, and non-repudiation [Buc16]. No single cryptographic operation may ensure all properties at the same time. Instead, multiple operations must be combined to achieve this end.

Encryption

Encryption ensures the confidentiality of data, that is, it guarantees that only certain parties may access and read it [Buc16]. To this end, it transforms the original data—the *plaintext*—such that it is no longer distinguishable from random data. This result of an encryption is a *ciphertext*. In order to gain back the plaintext from the ciphertext, the backtransformation *decryption* can be applied. In order to perform en- and decryption deterministically, one requires at least one cryptographic key. Several types of encryption algorithms—or *ciphers*—are distinguished based on the number of keys involved.

Symmetric Encryption Symmetric or secret-key ciphers use the same key for both en- and decryption. This setup requires to either store the key between en- and decryption or to transmit it from the encrypting to the decrypting party. Since encryption aims at ensuring confidentiality of data, there often is no easily available, secure alternative way to transmit or store keys. If a channel were available, it could have instead been used to transmit the plaintext to begin with. The more parties are to be involved, the more aggravated this issue becomes. As a

result, the rather complicated distribution and management of keys is the major drawback of symmetric encryption. Symmetric ciphers may further be distinguished into stream and block ciphers [Buc16]. Stream ciphers encrypt plaintext bit by bit by transforming the key into a key stream. Currently, no stream ciphers are recommended for use [GGF17, fISB17]. In contrast, block ciphers split the plaintext into blocks of equal length. Subsequently, these blocks are encrypted into blocks of ciphertext with the same size. The most widely accepted block cipher is *AES* [DR98, DR99, Buc16].

For the purpose of this thesis, we look further into block ciphers. Block ciphers require further configuration along two dimensions. There are first several different options for how individual plaintext blocks are encrypted and ciphertext blocks are re-assembled into a complete ciphertext. How a block cipher handles individual blocks is determined by its *block cipher mode of operation*. The simplest of these modes is the *Electronic Codebook (ECB)* mode [Buc16]. When following ECB mode, a block cipher encrypts each plaintext block on its own and subsequently concatenates the ciphertext blocks independently of the other blocks. This approach, however, results in the same plaintext blocks being encrypted into the same ciphertext blocks, therefore revealing patterns in plaintexts and leaving the ciphertext insecure [GGF17, fISB17]. Various other modes of operation such as Cipher Block Chaining (CBC) [EMST78, Buc16], Counter Mode (CTR) [DH79, Buc16], Galois/Counter Mode (GCM) [MV04], or Output Feedback (OFB) [Buc16] have been suggested to address this shortcoming. Common to all alternatives are two things of relevance for this thesis. First, the encryption of one plaintext blocks is influenced by encryption of another, usually by XOR-ing the current plaintext block with the previous ciphertext blocks before encrypting it. Second, they require an *initialization vector (IV)* on top of key and plaintext that either needs to be *random* (CBC, CFB, OFB) or *unique* for each plaintext (GCM, CTR) [Buc16]. The second configuration parameter of block ciphers revolves around block and plaintext lengths. Each block cipher has a fixed block size, that is, they require each plaintext block to have a certain size. Yet, real-world plaintexts are unlikely to always be of a length that is divisible by a block cipher's block size. In consequence, the final plaintext block may end up short in most real-world scenarios. To encrypt plaintexts of arbitrary lengths, block ciphers pad the last block of each plaintext to their block size. The most commonly used padding schemes for block ciphers *PKCS5* and *PKCS7* [Hou09] append the number of bytes, which need to be added, as many times as are missing from a complete block. In other words, a block cipher following PKCS5/PKCS7 padding adds twenty bytes with the value 20 to a final block that is missing twenty bytes.

Asymmetric Encryption In asymmetric or public-key encryption, two keys are involved. One of them should be kept *private*, the other one is considered *public* [Buc16]. Asymmetric ciphers encrypt the plaintext with the public key that belongs to the party whom the data is for. They, in turn, may decrypt the ciphertext using their own private key. This setup gets around the problem of key distribution symmetric encryption exhibits because the public key can be shared freely. The practicality of asymmetric ciphers is significantly reduced by their comparatively low performance. While symmetric ciphers rely on simple shifting and permutation operations, the security of asymmetric ciphers is largely based on the hardness of their underlying mathematical operations (e.g., prime factorization [RSA78]), making them generally more computationally expensive than symmetric ciphers. Often used algorithms include RSA [RSA78, Buc16, fISB17] and ElGamal [Gam84, Buc16].

Hybrid Encryption To achieve the best of both worlds, symmetric and asymmetric ciphers are often combined in practice into hybrid approaches. In such approaches, the asymmetric cipher encrypts the cryptographic key that is employed by the symmetric cipher. The symmetric

cipher's role is to encrypt the actual data. Through this setup, the main drawbacks of both symmetric and asymmetric encryption get mitigated. First, thanks to the asymmetric encryption of the symmetric key, this key does not need to be shared. Instead, only the parties' public keys require sharing. Second, the biggest portion of the plaintext is encrypted through the faster symmetric cipher. As a result, hybrid approaches achieve better performance than asymmetric ones on their own.

Hashing

Hashing is a cryptographic operation aiming to ensure integrity of the data it is performed upon [Buc16]. Some data's integrity is ensured, when it is guaranteed that this data has not been modified unnoticeably or tampered with. To ensure this guarantee, a hashing operation maps input data of arbitrary length onto data of a fixed length, often referred to as the *hash*. In order for such an operation to be secure, the hashing algorithm has to be a) *irreversible* and b) free of *collisions* [Buc16]. If the former condition is violated and there is a back-transformation, the hash would reveal the original data. The latter condition is broken if a hash algorithm produces the same output for two different inputs. No real-world hash algorithm can provably attain both properties [Buc16]. Instead, for practicality reasons, they are approximated: It must be *practically* impossible to retrieve the original data from a hash, find an input that produces the same hash as a given input, and, lastly, compute the same hash for two different inputs [fISB17]. The algorithm families SHA-2 and SHA-3 are most commonly recommended to use [fISB17, GGF17].

In practice, hashes are most often used to provide means for secure password storage for log-in systems. Since secure hashing algorithms ensure integrity and do not allow for the original data to be computed from a hash, only password hashes must be stored, not the passwords themselves. When a user attempts to log into their account, the password they enter is hashed in the same way and the two hashes can be compared. To circumvent this security measure, attackers may store the hash values of common passwords in look-up tables. One mitigation of this line of attack has data be appended to the passwords before hashing. This data — also called *salts* — needs to be *random* and can be stored alongside the password hash. The purpose of salts is not to introduce an unknowable factor to the hashing process, which is why they do not need extra protection wherever they are stored. Instead, the random appendix to each password causes no password to be common anymore, significantly limiting the benefits of lookup tables.

Message Authentication Codes (MAC)

Message Authentication Codes (MAC) are closely related to hashing operations, but additionally involve one cryptographic key [Buc16]. Thanks to this key, a MAC may not only ensure the integrity but also the *authenticity* of the mac-ed data. In practice, MACs are employed when a sender of a message intends to prove they are the party who has sent a message. The MAC algorithm computes a tag using a secret key. They then send message, key, and tag to the receiver. The key has to be kept secret during transmission. When the receiver wants to check the authenticity, they apply the same MAC algorithm and key to the message. If the tag they create on their end matches the one they have received from the sender, the message indeed comes from the sender [Buc16]. Similarly to symmetric-key encryption, the main drawback of MACs relates to key distribution and management. The confidentiality of the key needs to be ensured at all times. Once the key is compromised, the receiver can no longer be sure that the message was not sent by someone else. Commonly, MAC algorithms are based on hashing algorithms, often then called keyed-hash MACs or HMACs. Alternatively, block ciphers may

also be used to compute MACs. MAC algorithms based on block ciphers are called CMAC. There is also a MAC algorithm that is based on the mode of operation GCM called GMAC.

Digital Signatures

Digital signatures provide the public-key alternative for data integrity to the secret-key approach of MACs, that is, two keys are involved in computing and verifying digital signatures [Buc16]. Each party again has a private key, they need to keep secret, and a public one, they can share openly with everyone. In contrast to asymmetric encryption, the signature algorithm *signs* the input data based on the signing party's private key. The data, along with the signature, may then be shared with other parties. These, in turn, may *verify* the signature using the signing party's public key. On top of *integrity* and *authenticity* of the data, this approach further guarantees *non-repudiation* of the sender [Buc16]. Since public keys are public, everyone can verify their signature on a piece of data, leaving them no option to claim the signature did not originate from them.

2.1.2 Transport Layer Security (TLS)

Transport Layer Security (TLS) is a cryptographic protocol for secure communication over digital channels [Res18, fISB19]. TLS is most commonly used on the internet to secure communication between browsers and web servers. The protocol has originally been suggested under the name Secure Socket Layer (SSL) protocol and has been released in seven versions (SSL 1.0, 2.0, 3.0 and TLS 1.0, 1.1, 1.2, 1.3). Currently, all TLS versions below TLS 1.2 are considered insecure [fISB19]. For all less recent versions, practical attacks that break security guarantees of the protocol exist.

When a client and a server attempt to establish a connection, they perform a *handshake* [Res18]. In this handshake, they agree on the security parameters of their communication, that is, TLS protocol as well as the *cipher suite* used. A cipher suite is a set of three types of cryptographic algorithms: key agreement, symmetric cipher, and MAC. By combining these three types of algorithms, TLS-secured communications achieve a wide array of security properties. First, before any messages containing payload may be sent, client and server must agree on a key. During the key-agreement process, the two parties may also check each other's identity. Second, client and server encrypt and compute a MAC for each message they send to each other. This way, the channel also ensures confidentiality and integrity of messages. Once the two communication partners have completed the handshake, they can send messages to each other over the TLS channel.

2.1.3 Implementation in Java

Java Security covers several areas, two of which are of relevance to this thesis: low-level cryptography and secure channels. For both areas, the Java Development Kit (JDK) not only ships with a set of APIs, but also with a default implementation of these interfaces. Both APIs follow a *provider* model. That is, they define interfaces separate from implementations and enable other implementations (hereafter referred to by their official name *providers*) to be plugged into the interfaces. This design facilitates a greater flexibility in terms of which algorithms are implemented as well as which implementation of a given algorithm is used. In the following, we discuss the two APIs in more detail.

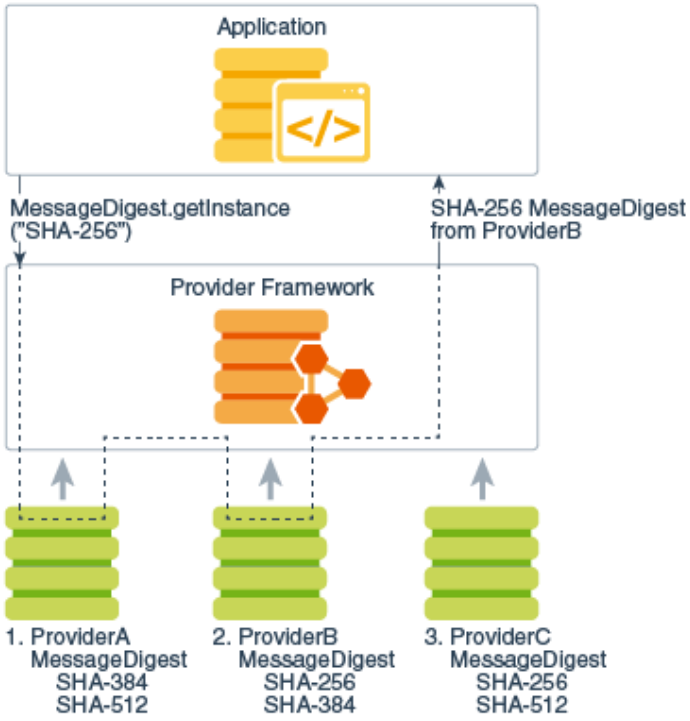

Figure 2.1: Provider-infrastructure-workflow Visualization from JCA Reference Guide [Inc17]

Java Cryptography Architecture (JCA)

The Java Cryptography Architecture (JCA) [Inc17] provides interfaces for low-level cryptographic algorithms. Its interfaces cover a wide range of cryptographic services, including symmetric and asymmetric encryption, hashing, MACs, digital signatures, key distribution, and management. The JCA, bootstrapped with the JDK default implementation, is also the most widely used Java Crypto API [NKMB16]. The main default provider is the SunJCE provider [Ora20a]. One other often-used provider is BouncyCastle [Leg18]. BouncyCastle may be used both independently through its own API as well as through the JCA interfaces as a provider.

We take the example from the official JCA documentation [Inc17] to illustrate idea behind the provider architecture. We show a call to `MessageDigest.getInstance("SHA-256")` in Figure 2.2 and how this call is handled by the JCA in Figure 2.1. During the execution of a Java program, the JCA maintains a list of *providers*. This list contains providers that have registered with the JCA. All providers in that list offer cryptographic services to the Java program if it uses the JCA. When a program makes a call like the one to `MessageDigest.getInstance("SHA-256")`, the JCA iterates through the providers in the list and requests the algorithm SHA-256 from each one after another. The JCA selects the implementation of the first provider that confirms it supports the requested algorithm. When no provider supports an algorithm, the JCA throws an `UnsupportedAlgorithmException`.

```
23 public static void main(String... args) {
24   MessageDigest md = MessageDigest.getInstance("SHA-256");
25 }
```

Figure 2.2: A Call to `MessageDigest.getInstance("SHA-256")`

Many APIs in the JCA contain this static factory method `getInstance()`. Or rather: several overloads thereof (e.g., `Cipher`, `MessageDigest`, `MAC`, `KeyGenerator`). The first parameter in each `getInstance()` method is the requested algorithm. However, programs may provide a provider as the second parameter. Doing so allows them to request the implementation of specific providers.

Java Secure Socket Extension (JSSE)

The Java Secure Socket Extension (JSSE) [Ora19a] provides security services revolving around TLS. As with the JCA, the JDK not only ships with the APIs of the JSSE, but also comes with an implementation, i.e., a provider. The default provider is the SUNJSSE provider [Ora20b].

The JSSE provides APIs to establish TLS sockets. When two sockets connect to each other through a TLS connection, they offer an `OutputStream` and an `InputStream` to send and receive data, respectively. When one socket sends data for the first time, the two sockets automatically perform the TLS handshake. The JSSE provides a default configuration for TLS sockets. However, both the list of enabled cipher suites as well as the list of enabled TLS protocols contain insecure entries [Ora20b, fISB19]. Fortunately, one can configure cipher suites and TLS protocols manually by calls to `setEnabledCipherSuites()` and `setEnabledTLSProtocols()`, respectively.

2.2 Static Data-Flow Analysis

A static data-flow analysis is a type of program analysis that tracks data through a program to reason about the program without executing it. Doing so enables an analysis to determine certain properties about the program. Originally employed for compiler optimization, such analyses are nowadays more often used to identify security vulnerabilities.

Static data-flow analyses usually transform a program under analysis into an intermediate representation (IR). IRs generally restrict themselves to a low number of language constructs an analysis must cover in order to analyse a program. Similarly, some IRs follow *three-address code*, that is, they restrict the number of operands per statements to three. This limitation may require to split nested statements in the original language into several simpler ones, but still allows the IR to express all language elements. Such restrictions simplify analyses because they significantly reduce the number of cases an analysis must cover. When all types of loops are translated to the same (set of) IR constructs, there is no need for an analysis to distinguish between these types of loops.

Subsequently, analysis tools parse the code in IR into data structures that facilitate further analysis. To capture the control flow of individual methods, analyses employ control-flow graphs (CFGs) that represent individual statements as nodes and potential control-flows between them as edges. Analyses residing on the level of individual methods are called intraprocedural. Analyses that go beyond the boundaries of individual methods by taking into account the effects of method calls are interprocedural. To model effects of method calls, caller-callee relationships are recorded in call graphs. Call graphs model methods as nodes and a call from one to another as an edge between their corresponding nodes. To represent a whole program, analyses sometimes combine both individual CFGs and the program's call graph to an interprocedural control-flow graph (ICFG).

2.2.1 Types of Analyses

Several types of data-flow analysis are used for security purposes, most notably taint analyses for detecting SQL injections. For the purpose of this thesis, we focus our attention on the two

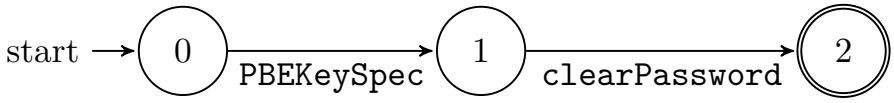

Figure 2.3: State Machine for Class `PBEKeySpec` (Without the Implicit Error State).

types of static data-flow analysis that are of relevance for it: typestate and points-to analysis.

Typestate Analysis

A typestate of a given object specifies the operations that may be performed on this object at a given point in the program [SY86]. A typestate analysis may be used to determine whether operations on an object are performed in no other than the correct order. To this end, the analysis holds a usage protocol specification of the typestates of an object, usually in the form of a finite state machine. The target program is then checked by the typestate analysis for its compliance with this usage protocol. State machines are enriched with an error state that is transitioned to when the analysis detects an operation prohibited by the current typestate.

Consider the example of object `spec` of type `PBEKeySpec` from Figure 1.1. We display `PBEKeySpec's` state machine in Figure 2.3: one has to first invoke the constructor and, subsequently, method `clearPassword()`. We leave out the error state for simplicity's sake. For the invocation of the class' constructor in Line 3, a typestate analysis transitions the state machine from state 0 to state 1. If method `generateKey()` contained a call to `clearPassword()`, it would further transition from state 1 to state 2. The latter state is the only accepting state of the state machine as only after `clearPassword()` is called, the object has been used correctly. However, this method is never invoked on `spec` during its lifetime (i.e., until the end of `generateKey()`). As a result, the typestate analysis reports a violation of the usage protocol, still being in state 1. Similarly, if the constructor invocation were followed by a call to a method different from `clearPassword()`, the analysis would transition the state machine to the error state.

Points-to analysis

A points-to analysis computes allocation sites of variables in a given target program. For a program variable, such an analysis compiles a set of allocation sites—also called a points-to set—this variable can point to [SB15]. Such information may be used to derive alias information for objects and simplifies most other types of more complex analyses. Points-to analyses hence often serve as auxiliary analyses for other analyses.

Consider the typestate analysis from above as an example. Assume that we defined a new local variable `specAlias` of type `PBEKeySpec` as an alias of `spec` in Line 4 in `generateKey()`. Assume further, we added a call to `specAlias.clearPassword()` at the end of the method. In order for the typestate analysis described above to determine that both `spec` and `specAlias` alias, it may compute the points-to sets of both. This computation yields that both variables can point to objects being created at the same allocation site, so possibly to the same object.

While computing points-to information is not the only way to determine aliases, the information supports the typestate analysis. The same analysis without awareness of aliases would treat `spec` and `specAlias` as two distinct objects and conclude that neither is used correctly. The former because `clearPassword()` is never called, the latter since no constructor was invoked.

2.2.2 Analysis Configuration

Program analyses may exhibit different sensitivies that impact their precision and scalability. In the following, we give a brief introduction to the ones relevant to this thesis.

Flow Sensitivity

Flow sensitivity refers to the ability of a static analysis to determine the order of statements and the general control flow of a method [SB15]. Most meaningful data-flow analyses are flow sensitive to be able to track how data *flows* through a program.

Context Sensitivity

An interprocedural analysis is context-sensitive when it keeps track of calling contexts while analyzing a method [SB15]. More specifically, that means, such an analysis is capable, after finishing the analysis of one method, of returning to the particular call-site that originally caused it to even go to that method.

Field Sensitivity

Field sensitivity refers to how an analysis treats fields. Data-flow analyses model field accesses through access paths. Such a path generally consists of an object in a base class and a chain of field accesses. The literature distinguishes three approaches of recognizing fields. *Field-insensitive* analyses do not take into account fields whatsoever. Instead, they approximate fields as their base object. An alternative to field-insensitive analyses that does not ignore fields are *field-based* approaches. Here, fields are modelled as the field's name and its declaring type. Such approaches *may* provide better precision, if the program under analysis does not contain many instances of objects of the same type. However, compared to field-insensitive approaches, they also lose information about the base object. Both approaches may hence be seen as orthogonal. Lastly, *field-sensitive* analyses include the base object in their modelling of a field. They thereby distinguish different instances of the same field and offer the most precise treatment of fields as a result. Field-sensitive approaches may however run into scalability issues because access paths may, by definition, be of infinite length. To approximate such paths, analyses have traditionally applied k-limiting approaches [Spä19], limiting the number of field accesses that are modelled to a given k. In more recent years, k-limiting has more and more been replaced by pushdown systems [RSJ03, Spä19].

Path Sensitivity

A static analysis is path-sensitive when it can distinguish between program paths. Only path-sensitive analyses can meaningfully reason about the full effects of language constructs that make a program branch. Path-sensitivity is, however, an undecidable problem statically. Whether or not an if-condition evaluates to true or false might depend on a multitude of factors that are hard or impossible to analyse (e.g., user input, other previous if-statements, the result of a loop). In cases where input from the outside is involved, there is no way at all for the analysis to determine which path is taken.

On-Demand vs. Whole-Program

Interprocedural analyses have traditionally been used to analyze the whole target program. This approach, however, is quite resource-intensive, causing many more complex analyses to take several hours to days and to consume large amounts of memory. In recent years, static analyses have increasingly been designed in an on-demand manner [Bod18, SDAB16a, SAB17, RXA+19]. In contrast to whole-program analyses, on-demand analyses only analyze what is explicitly necessary. A demand-driven alias analysis, for instance, does not compute all aliases of all variables, but only answers an alias query for one particular set of variables at the given program

point. Jaiswal et al. [JKC18] argue that on-demand analyses not only make the analysis more scalable, but simultaneously also more precise.

3 CogniCrypt

Chapter 1 illustrated common pitfalls when using Crypto APIs, discussed these APIs' widespread misuse as a consequence, and motivated our work on fixing this misuse. This chapter introduces CogniCrypt, our solution to the misuse and the main contribution of this thesis. CogniCrypt integrates several types of tool support for Crypto APIs into one unified tool. The support application developers receive from CogniCrypt enables them to use the APIs without first needing to understand how to correctly and securely use them.

We have prototypically implemented CogniCrypt as an Eclipse plugin that supports Java developers. As of now, the plugin comprises the two types of tool support we contribute as part of this thesis:

- CogniCrypt$_{\text{GEN}}$– Generate secure implementations for common use cases that involve cryptography (e.g., data encryption).
- CogniCrypt$_{\text{SAST}}$– Analyze developer code and generate warning messages for misuses of cryptographic APIs.

A first outline of this research has been given by Arzt et al. [ANA+15] as early as October 2015. In the remainder of this chapter, we will give a high-level description of how a user would interact with CogniCrypt as a plugin and describe the use cases and APIs the plugin supports. While the plugin's support is limited to Java, the concepts we lay out over the course of this thesis are applicable to other programming languages as well.

3.1 CogniCrypt in a Nutshell

The Eclipse plugin CogniCrypt supports its users — Java application developers — in securely using Java Crypto APIs. In particular, through the code generator CogniCrypt$_{\text{GEN}}$, CogniCrypt can support them in implementing PBE as we described it in Chapter 1. To generate code implementing this use case, a user has to click on the CogniCrypt code-generation button in the Eclipse tool bar. The dialogue shown in Figure 3.1 pops up and the user has to select the first icon from the list of use cases on the left. While the icons should give users a hint regarding what kind of use cases CogniCrypt supports, CogniCrypt provides a more detailed description of the use case that is selected at any given time. The user then answers a few high-level questions that do not require deep cryptography knowledge. The answers to these questions help CogniCrypt generate the appropriate source code. One such question for PBE is "Which method of communication would you prefer to use for key exchange?" as

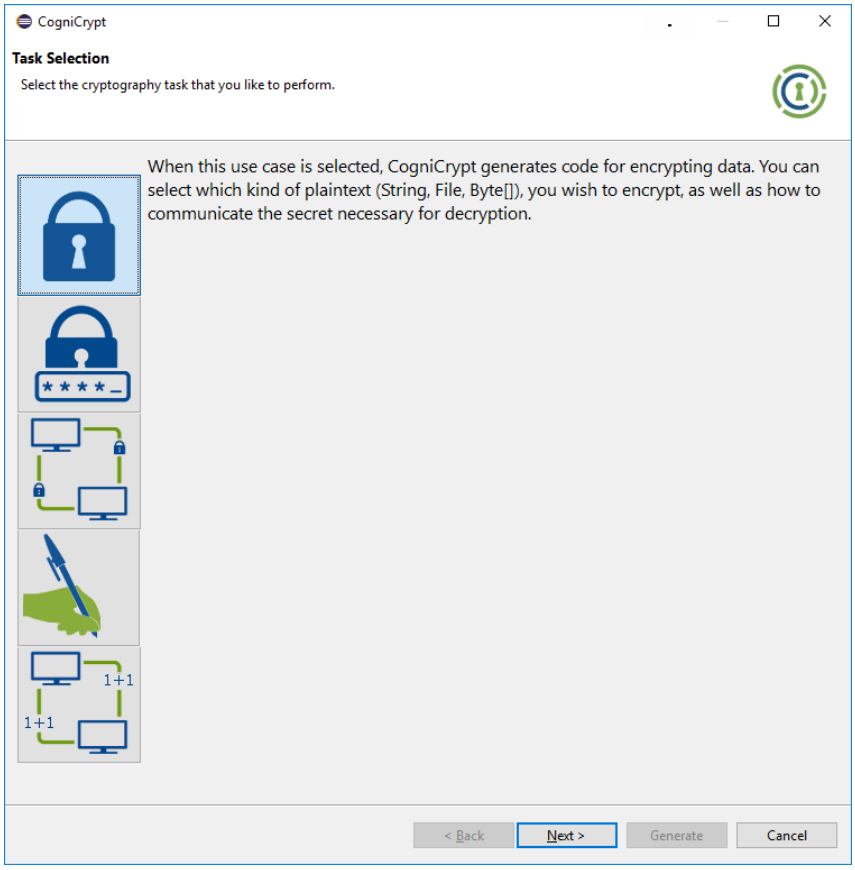

Figure 3.1: Dialog for Task Selection.

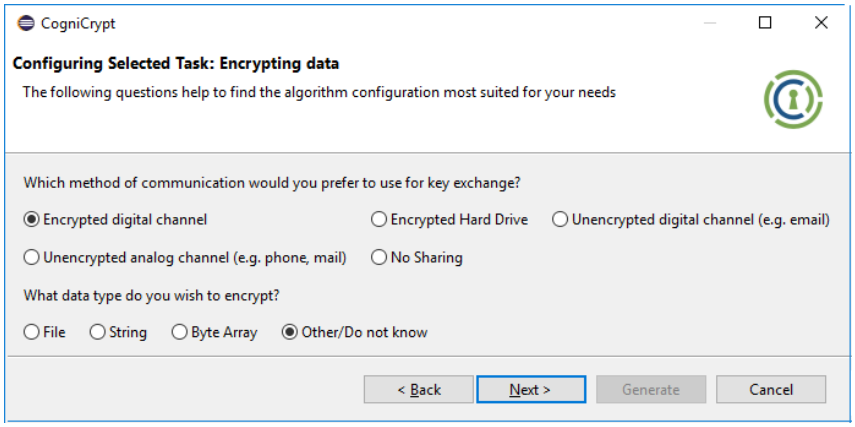

Figure 3.2: Questions for PBE.

Figure 3.2 shows. Once the user has answered all questions, they have to select a file as depicted in Figure 3.3. After they hit the Generate button, CogniCrypt generates two code artefacts. First, it generates code that implements PBE into the package `de.cognicrypt.crypto`. Second, CogniCrypt also generates a method `templateUsage()` into the file the user selected. This method showcases how the implementation for PBE can be integrated into the user's project.

In addition to CogniCrypt$_{\text{GEN}}$, CogniCrypt notifies the user of misuses of Crypto APIs by running the static analysis CogniCrypt$_{\text{SAST}}$. Depending on how CogniCrypt is configured, CogniCrypt$_{\text{SAST}}$ is either run automatically every time the code is compiled or must be triggered

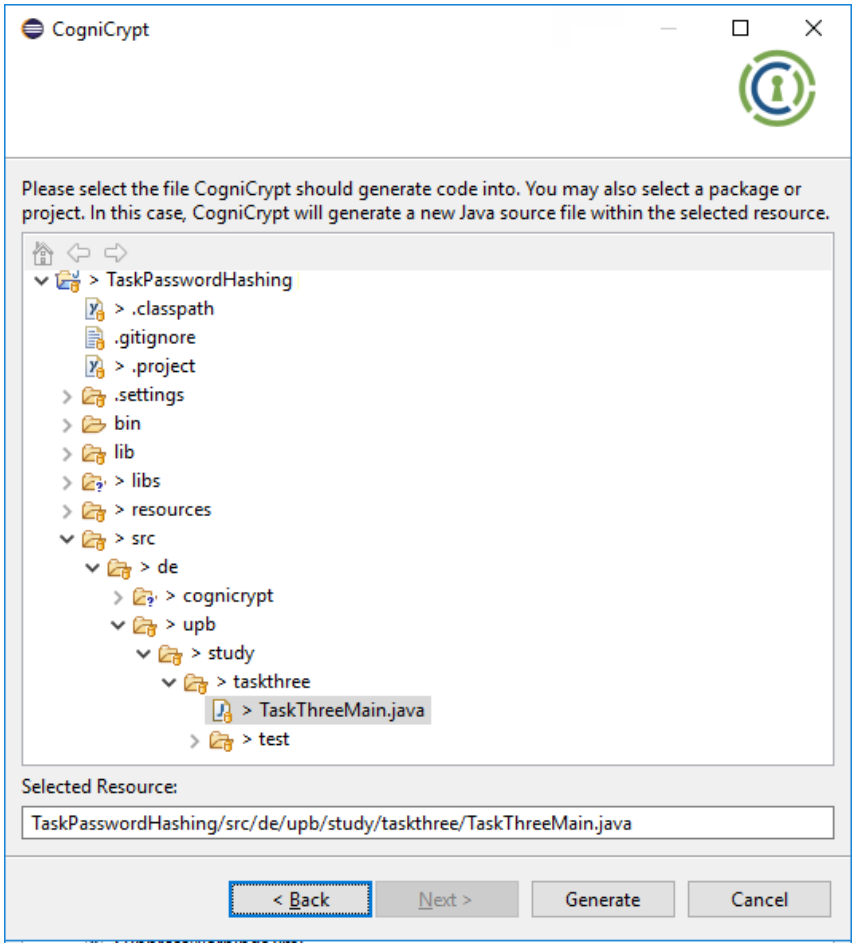

Figure 3.3: Selecting the File into Which `templateUsage()` is Generated.

explicitly by clicking the CogniCrypt analysis button in the Eclipse tool bar. The analysis ensures that all usages of cryptographic APIs remain secure, even when the developer modifies the generated code for better integration into their project or to add some functionality. Moreover, if the developer uses the cryptographic APIs directly (i.e., without using CogniCrypt$_{\text{GEN}}$), running CogniCrypt$_{\text{SAST}}$ ensures secure usage of the APIs. CogniCrypt generates an Eclipse error marker for each detected misuse of the supported cryptographic APIs. Figure 3.4 depicts a warning issued by CogniCrypt when the user changes the generated code for PBE to use the insecure ECB mode.

3.2 Integrated Components

CogniCrypt$_{\text{GEN}}$ provides support for Crypto APIs in a use-case-based manner. In this section, we first discuss the use cases that CogniCrypt as a plugin currently supports, providing short conceptual descriptions, how they are implemented, and the decisions users have to make in the tool. We furthermore discuss the APIs CogniCrypt supports through CogniCrypt$_{\text{SAST}}$.

Figure 3.4: Error Marker by CogniCrypt when ECB Mode is Used.

3.2.1 Use Cases

Data Encryption

- Description: Encryption of data as string, byte array, or file. Implemented either through PBE or hybrid encryption.
- Implementation: Implementations of symmetric block ciphers in SunJCE Provider [Ora20a] such as AES.
- User Decisions: CogniCrypt asks the user which method of key exchange they prefer and which type of data their application should be able to encrypt.

Password Storage

- Description: Transformation of passwords such that they can be securely stored (i.e., hashing and salting).
- Implementation: Implementations of key derivation functions in SunJCE Provider [Ora20a] such as PBKDF2.
- User Decisions: CogniCrypt does not need any configuration for this task.

Secure Communication

- Description: A cryptographic channel based on the Transport Layer Security (TLS) protocol [Res18] for securely transporting data from one endpoint to another. The channel ensures confidentiality and integrity of the communicated data as well as authenticity of the communication partners.
- Implementation: Based on the Java TLS implementation in the Java Secure Socket Extension (JSSE) [Ora19a].
- User Decisions: CogniCrypt first asks the user whether they wish to implement the client or the server side of a connection, requesting the corresponding internet-address. It further requests to know which protocol is used for the channel (HTTPS vs. plain TLS). CogniCrypt then allows the user to select the desired security level, providing a safe default option for optimal cryptographic protection. In particular, CogniCrypt

disables insecure cryptographic parameters (i.e., cipher suites & TLS protocols) as some are turned on by default. This feature is crucial because TLS has a vast number of parameter choices, and, in principle, allows to configure insecure cipher suites that, for example, omit encryption or enable known attacks like RC4 weaknesses [ABP+13].

Signing Data

- Description: Digitally signing data and verifying existing signatures.
- Implementation: Implementations of signature algorithms in SunJCE Provider [Ora20a] such as SHA256withECDSA.
- User Decisions: CogniCrypt does not need any configuration for this task.

Secure Long-Term Storage

- Description: MoPS [WDV+17] ensures the integrity and authenticity of documents over long periods of time, since classical protection schemes (e.g., digital signatures) do not provide everlasting security. MoPS allows users to create customized long-term protection schemes by combining reusable components extracted from other existing solutions, improving performance and gaining flexibility.
- Implementation: The reference implementation of MoPS by Weinert et al. [WDV+17] has a RESTful API for configuring and maintaining file collections on remote systems. Using the API without proper guidance may lead to a configuration that uses outdated cryptographic primitives (e.g., SHA-1), performs poorly due to improper component selection, or relies on inappropriate trust assumptions.
- User Decisions: CogniCrypt asks the user at most four high-level questions (e.g., "Do you plan to add new files to your collection frequently?"). These questions identify the required features and the trust assumptions the user is willing to make. It then translates the user choices into the most suitable component selection based on the recommendations of Weinert et al. [WDV+17]. Finally, CogniCrypt generates glue code to configure the MoPS system accordingly and provide methods for securely storing files in the system.

Secure Multi-Party Computation

- Description: ABY [DSZ15] is a framework for mixed-protocol secure two-party computation (STC). It allows two parties to apply a function to their private inputs and reveal nothing but the output of the computation. ABY enables developers to implement STC applications by offering abstractions from the underlying protocols. Furthermore, ABY can securely convert between different protocol types, improving efficiency.
- Implementation: ABY is written in C/C++ to achieve high efficiency for the underlying primitives (bit operations, symmetric encryption) and has been encapsulated in Java Native Interface (JNI) wrappers to be used by CogniCrypt.
- User Decisions: CogniCrypt offers the user several STC example applications, e.g., computing the Euclidean Distance between private coordinates. The user can select different properties, depending on the deployment scenario. In the future, we plan to integrate custom applications as well.

3.2.2 APIs

CogniCrypt currently supports finding misuses of the JCA [Inc17], the JSSE [Ora19a], BouncyCastle both as a provider and through its own lightweight API [Leg18], as well as Google Tink [Goo]. The rules for the JCA have been developed by us during the development of CrySL. Rules for the JSSE and both BouncyCastle APIs have been developed by grad students in their job as student assistants with CogniCrypt. Lastly, the Google Tink rules have been developed by Rodrigo Bonifácio during his research stay at Paderborn University. Support for further APIs can easily be added through the use of CogniCrypt's specification language CrySL (Chapter 5).

3.3 Conclusion

In this chapter, we have given a high-level overview on CogniCrypt as an approach and its prototypical implementation as an Eclipse plugin. CogniCrypt enables developers to securely integrate cryptographic components into their Java projects, especially if they have little experience with cryptography. CogniCrypt smoothly integrates into a developer's workflow to generate secure code for cryptographic use cases and detect misuses of Crypto APIs in their code. The tool is publicly available[1] and open-source.[2]

In the following chapters, we will discuss the underlying concepts and evaluations of the involved individual components, that is CogniCrypt$_{\text{SAST}}$ (Chapter 6) and CogniCrypt$_{\text{GEN}}$ (Chapter 7) in more detail. In Chapter 8, we present an empirical evaluation of the tool. First, however, we consider other work addressing the same or similar problems as CogniCrypt does in Section 4.

[1] `www.cognicrypt.org`
[2] `www.github.com/eclipse-cognicrypt/cognicrypt`

4 Related Work

In this chapter, we discuss related work that aims at fixing misuse of cryptographic APIs. We leave the work more closely related to CogniCrypt$_{GEN}$ and CogniCrypt$_{SAST}$, i.e., work on code generation and static program analysis to fix cryptographic misuse, to Sections 6.6 and 7.4, respectively. We are not aware of any integrated approach that addresses cryptographic misuse as systematically as CogniCrypt. However, multiple directions have been explored.

4.1 Usability & Re-design of Crypto APIs

In the discussion of Security- and Crypto-API usability, Wurster and van Oorschot [WvO08] and Green and Smith [GS16] center the perspective of the application developer, i.e., the API's user. Wurster and van Oorschot [WvO08] argue that, due to division of labour in contemporary software development, the average developer does not have sufficient knowledge to be trusted with (low-level) Security APIs. To address this issue, they suggest enhancing Security APIs to limit the decision space of their users and make the APIs more usable to them in the process. The often-given suggestion to train developers better, they argue, is insufficient. Instead, developers should be incentivized on multiple levels to use secure options, including through usable APIs. Green and Smith [GS16] agree with the general sentiment of not expecting the average developer to be a security or cryptography expert. They argue that especially Crypto APIs that usually require a remarkable level of cryptographic expertise need undergo a re-design. To this end, they suggest ten design principles, ranging from close-to-code-level recommendations like secure defaults and misuse causing visible errors to more abstract ideas such as integrating cryptographic functionalities into non-cryptographic (i.e., use-case-oriented) APIs or the introduction of a testing mode. Their suggestions have shaped the design processes of a lot of recent Crypto APIs and usability discussions around them [PHR19, MKW18].

Iacono and Gorski [IG17] introduce some nuance to the discussion by distinguishing two types of Security APIs. On the one hand, there are *Security Primitives APIs* that enable their users to select individual algorithms and configure them. On the other hand, they classify more high-level task-based APIs as *Security Controls APIs*. Such APIs do not expose cryptographic algorithms directly, but instead provide more high-level use-case oriented interfaces. They survey software developers about their experience with Security APIs. From the results, they conclude that there is a need for more *Security Controls APIs*. This is because most existing security (and therefore cryptographic) APIs classify as *Security Primitives APIs* and will likely always remain a playground for those with security expertise, but not reach beyond. This conclusion

has reached the state of a consensus in the literature on the usability of Crypto APIs [PFZ17, IKND16, GS16, NKMB16, ABF+17, KLCL18, vdLRWW18, PHR19] and we share the sentiment as well. We aim to address the issue not by designing a new API, but through CogniCrypt.

Pierzu et al. [PFZ17] take a different approach by arguing that the problem of Crypto-API misuse may be mapped onto the problem of the symmetry of ignorance. An application developer is an expert in the domain of their respective application (e.g., databases, networking), but not (necessarily) in cryptography. Similarly, a cryptographer designing and implementing a cryptographic API is unaware of the needs and wants of application developers as their domains lie outside their own expertise.

Van Linden et al. [vdLRWW18] suggest to take a step back from designing new APIs and instead first determine a suitable abstraction level for them. They argue, the research community should first find appropriate metaphors for cryptographic operations that are understood by developers. Following this notion, they lay out a research agenda, according to which they plan to conduct an empirical study to come up with new metaphors. Those, they argue, could make up the user interface for tools such as CogniCrypt's code generator.

Acar et al. [ABF+17], Mindermann et al. [MKW18], Patnaik et al. [PHR19], Oliveira et al. [OLR+18], and Nadi et al. [NKMB16] have all empirically investigated the usability of Cryptographic APIs in Python, Rust, and Java, respectively while largely coming to similar conclusions. Acar et al. [ABF+17] conduct an empirical study, in which they ask 256 Python developers to implement cryptographic use cases with one of five Crypto APIs. Among the sample set of crypto libraries, there are both low-level as well as more use-case-oriented libraries. They assess functionality and security of participants' code through manual inspection. The authors find that task-based APIs do lead to better security due to the reduction of decision space for API users, but only slightly. Whether or not an API offered good documentation and readily usable code examples better predicts the security of the code participants produce. Similarly, supporting a broad range of cryptographic operations such as key management also improves application-code security. Consequently, they demand from library designers to not only focus on simplifying APIs, but to also prioritize documentation and providing a full feature set.

Mindermann et al. [MKW18] evaluate two Crypto APIs for Rust in terms of their usability by having 22 students implement one encryption-related task with either. As APIs, they have chosen one low-level and one higher-level one. Somewhat surprisingly at first glance, participants with the higher-level API produce less functional code than with the low-level API with a mixed picture concerning security. However, their study exhibits two crucial threats to validity. First, despite randomly assigning participants to a library, they do not end up distributed equally in terms of experience level. Second, even more importantly, the documentation of the low-level API contains an example that exactly fits the use case participants are asked to implement. The authors show that participants who have not used the code example during the study do not produce more functional code than participants using the more high-level API. On these grounds, the authors demand to link to an API's documentation prominently and help with making configuration choices among other things, but do not advise against high-level APIs.

Oliveira et al. [OLR+18] examine the role of API blindspots. By blindspot, they are referring to the possibility of using an API in a functionally correct manner, but that still exposes an application containing this use to vulnerabilities. They do not exclusively target Crypto APIs, but include several cryptographic examples in their work. To measure the impact such blindspots have, they conduct an online study with 109 developers. Participants are asked to solve six short programming tasks, four of which use APIs with a blindspot. Subsequently, they need to answer questions about the security of the previously worked-on programming task. Oliveira et al. [OLR+18] find participants are significantly less likely to correctly solve the programming task, if the used API contains a blindspot. As a result, they recommend removing blindspots

from legacy APIs and pilot testing new ones before release to detect blindspots early on.

Nadi et al. [NKMB16] home in on the usability of Java Crypto APIs. They triangulate the results of four separate studies. First, they investigate the top 100 posts about Java cryptography on a popular Q&A platform. Second, they survey all users of that platform who have posted on the platform under said topic in two studies. Lastly, they check 100 randomly Java Github that use the JCA to determine what developers are actually attempting to implement when using the JCA. Their results strongly suggest that developers generally agonize over using Crypto APIs in Java, overwhelmingly the JCA. According to survey participants' self-reporting, at least 81% struggle at least occasionally with identifying the correct call-sequence, 75% with identifying the parameters and 56% with identifying the right crypto concepts. The results support previous arguments that application developers are not necessarily cryptography experts. However, they do also underline that Crypto APIs exposing algorithms to their users are too low-level. Survey participants further request help first and foremost through re-designing APIs, but also by providing better documentation and tool support. New API designs should be more use-case-oriented and more high-level. Documentation must contain code examples. The tool support participants have in mind largely tests programs for their cryptographic correctness and provides implementations for common cryptographic use cases.

Panaik et al. [PHR19] map concrete violations of the design principles by Green and Smith [GS16] to usability smells in code that uses the Crypto API. To this end, they first investigate 2,400 posts about cryptographic libraries on a popular Q&A platform to come up with a taxonomy of usability smells. They recognize that Crypto APIs largely miss proper documentation with examples and guidance for the developer in the form of error messages in case of security issues. They further diagnose that Crypto APIs confuse their users causing them to prototype with different APIs trying to figure out which one is suitable under the given circumstances. This problem even extends to after the code has been written when it turns out the developer has used it incorrectly. As a second step, they investigate the prevalence of these smells in three commonly used Crypto libraries. The authors conclude that the three libraries to varying degree exhibit all smells they have identified.

Indela et al. [IKND16] propose a new design for Crypto APIs, incorporating a lot of suggestions from other work [GS16, NKMB16]. Their API offers high-level and use-case-based interfaces for regular software developers as well as more low-level ones to security engineers. The underlying library enforces the use of high-level interfaces through the mandatory tagging of all data going through it as sensitive or not. That is, any data that is marked as sensitive must be passed through one of the high-level interfaces. They also suggest cryptographic libraries to implement a Regulator design pattern that would enable them to regularly draw the latest recommendations on cryptographic algorithms from authorities like NIST [GGF17].

This line of research provides necessary steps to move forward. Only if Crypto-API developers learn from past mistakes, there is a chance the future brings progress. Since usability problems of cryptographic APIs have first been recognized, several attempts have been made at fixing them [BPF19, BLS12, FLW12, Lou, Pytb, Soda, Sodb, Mai, Pyta, Goo, Smi, PTSA$^+$, Cos]. Libraries that have been proposed as a result move in the right direction as they improve on past ones. They also serve as good objects for further usability research allowing the community to gain deeper insights on which advice actually works and which does not. However, as of now, many widely used Crypto APIs, especially for Java, are *Security Primitives APIs* with bad usability. While new libraries are designed and seek wider adoption, developers rather require more immediate support like CogniCrypt.

4.2 Propping up Libraries

Gorski et al. [GIW+] approach fixing Crypto-API misuse through improving security-related warnings and error messages from within the respective API. Their error messages alert to security-critical misuses of the API and concisely describe the risk involved, provide context, and offer options for fixes. To empirically evaluate their design, they implement their warnings for a common Crypto API in Python and conduct an experiment with 53 Python developers. Participants are asked to implement two programming tasks, either with the enhanced Crypto API or with the regular one. They find that 27% of submitted solutions of participants with the regular API to be secure, while the number jumps up to 51% with the enhanced API. They further observe that 73% of participants adopt suggestions from the security error messages.

As the empirical evaluation shows, the approach has the potential of improving the usability of existing APIs substantially. Other similar ideas could also be applied (e.g., libraries throwing exceptions when insecure configurations are used or deprecating of no-longer-secure algorithms). However, such approaches require to be adopted by API developers. If an API developer is not aware of them or chooses to not adopt them, the API user is on their own and needs to fall back to external support. CogniCrypt may provide such support.

4.3 Fixing Existing Resources for Helping Software Developers

When developers reach a road block during everday development, they often default to looking for code snippets on their search engine or Q&A platform of choice [BC14, TJVW19, ABF+16]. Incidentally, the latter are also appearing among the top results in the former. Past research has established that much of the code on such platforms is insecure [FBX+17, ABF+16, ASW+17]. As a result, developers seeking help on these platforms introduce vulnerabilities to their projects they are often not warned about as only few of them cause runtime exceptions to be thrown. More recent research takes up the task of improving the answers on a popular Q&A platform. Fischer et al. [FXK+19] discover secure alternatives for 99% of insecure code snippets implementing similar use cases. To detect these alternatives, they employ a learning approach using neural networks. The authors subsequently re-design the answer pages of the platform with insecure solutions to convince participants to rather go to threads with more secure answers. For evaluation, they conduct an experiment for which they have 27 students solve five cryptographic programming tasks. For two of the tasks, the solutions obtained with the re-designed pages provide significantly more secure code. In the case of the other three, the code is still more secure, but not enough to be statistically significant for varying reasons.

Huesmann et al. [HZH19] further the discussion of Acar et al.[ABF+17], Mindermann et al. [MKW18], Patnaik et al.[PHR19], and Nadi et al. [NKMB16] into what good documentation for Crypto APIs should look like. The authors conduct focus-group-style interviews with 26 seasoned developers. To each focus group, they show multiple forms of documentation and gather feedback on which ones are more desirable and why. Participants generally request a mix of different documentation styles. Winning out against everything else are code examples as they are requested by participants in all focus groups. On top of examples, participants also request the documentation to be well-structured, contain links to classical references of the code (e.g., JavaDoc) and provide knowledge on different levels of intricacy.

Mindermann and Wagner [MW18] contribute in a more direct and practical manner by providing JCA [Inc17] documentation in the form of code examples for the API's most commonly used use cases. Thereby, they implement a feature, strongly requested by participants of Nadi et al.'s survey [NKMB16] and Huesmann et al. [HZH19]'s focus groups. We discuss the examples again in our evaluation of CogniCrypt$_{GEN}$ in Section 7.3.

These three approaches provide practical support for application developers attempting to use low-level Crypto APIs. We view our work on CogniCrypt as complementary to all them. CogniCrypt provides more direct development support through CogniCrypt$_{SAST}$ and CogniCrypt$_{GEN}$ and also enhances API documentations through API specifications in CrySL. Through implementing our suggestion CogniCrypt$_{DOC}$ from Section 9.4, one could provide further documentation.

4.4 Security Awareness in Organisations

Prior work has also examined the lack of experience, awareness, and knowledge of application developers towards security topics, suggesting increasing awareness may bring about more secure code. In three different studies, Naiakshina et al. investigate whether students [NDT+17, NDTS18] and professional freelance developers [NDG+19] implement password-storage solutions securely. Regardless of test subject, they come to broadly the same conclusions. Developers largely do not appropriately address the security concerns such a programming task requires when not explicitly requested. To estimate whether salary plays a role in how security is addressed, they paid participants in their freelancer study different amounts of money ($100 vs. $200). The better-paid group does produce slightly, but not statistically significantly more secure code. However, they do find statistically significant differences between participants who are requested to pay attention to security and those who are not. These results, by and large, support claims that security often comes an afterthought and developers rather rather focus on functional correctness instead of security. To address this issue, the authors suggest explicitly prompting developers to care about security.

Tahaei et al. [TJVW19] conduct twenty semi-structured interviews with computer-science students, ranging from Bachelor to Ph.D. students with broad-branching career aspirations. They find that interviewees generally do not seem to have a cohesive and meaningful view of security, appearing to be significantly influenced by media portrayals of the topic. Similar to Naiakshina et al., they conclude that students' lack of security awareness significantly contributes to the widespread insecurity of real-world applications. Compared to Naikshina et al., Tahaei et al. suggest more far-reaching measures: They propose to synchronize software-engineering practices with what is being taught in academia.

Lopez et al. [LST+19] expand on this work through semi-structured interviews with seven professional application developers at one company. The interviewees generally agree that a) security is vital and they as individuals are responsible for it, b) security is taken seriously by their employer, c) their company is doing comparatively well, but d) that more can be done.

To a similar end, Haney et al. [HGT17] survey 121 developers from different organisations on their attitudes towards and practice around Crypto APIs. Their participants first echo earlier complaints about the usability of Crypto APIs. However, more than a third also mention having problems with justifying a focus on secure cryptography to their customers. Participants also report to rely heavily on standards such as FIPS [Lab19], but not use any testing tools to vet their own cryptographic code. In response to these results, the authors call for more usable testing tools and standards for cryptography.

Haney et al. [HTAP18] expand on this work by interviewing 21 professional security engineers. Participants report a high awareness for secure cryptography for both themselves and their respective companies. Companies make a point of achieving secure software products. Similar to Haney et al. [HGT17], participants declare a high reliance on cryptographic standards. Opinions on certifications and academic contributions differ. The former are seen as helpful to show to customers, but too expensive. The latter are often seen as too out-of-touch from the actual problems the industry face. They also put heavy emphasis on testing and an-

alyzing cryptographic code, but mention difficulties in doing so thoroughly and properly. The authors consequently demand from the research community to expand their focus to other populations and move away from solely individual solutions to what seem to be (also) organisational problems. The study does not target regular application developers directly, but instead security experts. One should thus be careful when drawing direct conclusions as to what needs to improved for the former group. However, learning more about the environment that produces security experts may shed light on how organisations might need to re-organise to achieve similar outcomes.

In none of the three studies [LST+19, HGT17, HTAP18] do the authors evaluate the security of the companies' products. However, the research by Tahai et al. [TJVW19] and Naiakshina et al. [NDT+17, NDTS18, NDG+19] suggests such a security-conscious environment may go a long way of producing secure software. While most of this work is concerned with security in general, its results seem applicable to the subdomain of cryptography as well. Adding to the suggestions made by Naiakshina et al. [NDT+17, NDTS18, NDG+19], Haney et al. [HGT17, HTAP18], and Tahaei et al. [TJVW19], we propose organisations implementing cryptographic software to integrate automated tool support such as CogniCrypt into their QA process. Furthermore, organisational practices are also of limited use to developers working alone or in very small teams. In such scenarios, our tool CogniCrypt may be of greater help.

4.5 Conclusion

In this chapter, we have discussed work related to CogniCrypt that is not focussed on program analysis or code generation. Most work does improve the situation either through suggesting new designs (Section 4.1), fixing weakspots in cryptographic libraries (Section 4.2), improving existing resources (Section 4.3), or providing measures to increase security awareness (Section 4.4). However, our discussion also reveals, there currently is no other approach that addresses cryptographic misuse as systematically as CogniCrypt does. Most work we discuss in this chapter may rather be seen as complementary to CogniCrypt.

5 CrySL

In this chapter, we present CrySL. CrySL is a specification language that enables cryptography experts to specify the secure usage of their Crypto APIs. CrySL serves as the foundation of CogniCrypt, although it remains invisible to users of the Eclipse plugin CogniCrypt'.

We designed CrySL for (and with the help of) cryptography experts. The language's lightweight special-purpose syntax is one result of that process. CrySL is meant to serve as a building block for different kinds of tool support, including documentation, patch generation, or use-case-based code generation as well as program analysis. Thanks to CrySL, addressing cryptographic misuse may go beyond methods that are useful for the general validation of API usage (e.g., typestate analysis [AAC+05, BA07, NL08, Bod10] and data-flow checks [ABB+09, ARF+14]) by enabling one to express domain-specific constraints related to cryptographic algorithms and their parameters.

5.1 Syntax

As we discuss in Section 5.5.2, mining API properties for Crypto APIs is extremely challenging, if possible at all, due to the overwhelming number of misuses one finds in actual applications. Hence, instead of relying on the security of existing usages and examples, we here follow an approach in which cryptography experts define correct API usages manually in a special-purpose language, CrySL. In this section, we give an overview of the CrySL syntax elements. A formal treatment of the CrySL semantics is presented in Section 5.2.

5.1.1 Design Decisions Behind CrySL

We designed CrySL specifically with crypto experts in mind, and in fact with the help of crypto experts. This work was carried out in the context of the large collaborative research center *CROSSING*[1] that involves more than a dozen research groups involved in cryptography research. As a result of the domain research conducted within this center, we made the following design decisions when designing CrySL.

White listing. During our domain analysis, we observed that, for the given Crypto APIs, there are many ways they can be misused, but only comparatively few that correspond to correct and secure usages. To obtain concise usage specifications, we decided to design CrySL to

[1] www.crossing.tu-darmstadt.de/crc_1119/

use white listing in most parts of a specification (i.e., defining secure uses explicitly, while implicitly assuming all deviations from this norm to be insecure).

Typestate and data flow. When reviewing potential misuses, we observed that many of them are related to data flows and typestate properties [SY86]. Such misuses occur because developers call the wrong methods on the API objects at hand, call them in an incorrect order or miss to call the methods entirely. Data-flow properties are important when reasoning about how certain data is being used (e.g., passwords, keys, or seed material).

String and integer constraints. In the crypto domain, string and integer parameters are ubiquitously used to select or parametrize specific cryptography algorithms. Strings are widely used, because they are easy to recognize, configure and exchange. However, specifying an incorrect string parameter may result in the selection of an insecure algorithm or algorithm combination. Many APIs also use strings for user credentials. Those credentials, passwords in particular, should not be hard-coded into the program's bytecode. A precise specification of correct crypto uses must therefore comprise constraints over string and integer parameters.

Tool-independent semantics. We equipped CrySL with a tool-independent semantics (Section 5.2). In the future, this semantics will enable us and others to build other or more effective tools for working with CrySL. On top of the two we present as part of this thesis — CogniCrypt$_{SAST}$ and CogniCrypt$_{GEN}$—, we discuss four more CrySL-based tools in Chapter 9. Furthermore, a dynamic checker to identify and mitigate CrySL violations at runtime is currently being developed.

Our desire to allow crypto experts to easily express secure crypto uses also precludes us from using existing generic definition languages such as Datalog or QL. Such languages, or minor extensions thereof, might have sufficient expressive power. However, following discussions with crypto developers, we had to acknowledge that they are often unfamiliar with those languages' concepts. CrySL thus deliberately only includes concepts familiar to those developers, hence supporting an easy understanding.

The resulting language is not, per se, limited to expressing usage constraints on Crypto APIs. While there are certain elements in CrySL, such as the integer and String constraints, that are more essential to cryptographic than to other APIs, we do assume the language to be capable of covering those other APIs as well. We nonetheless view CrySL as domain-specific because we tailored it to the domain of cryptography through our extensive domain analysis, which resulted in, among other things, the aforementioned language elements. We have, however, not conducted an in-depth investigation into CrySL's applicability to other APIs of other domains.

Rules in CrySL are split into multiple sections as a means to follow the separation-of-concerns paradigm. This way, required method calls are defined independently of forbidden ones, constraints on an object may be specified separately from assigning this object a role as method argument or return object of a method, and the correct order of method calls is defined without interference from object definitions or declarations of forbidden method calls. These separations improve readability and, as described further below, facilitate reuse of elements within a single rule. In early discussions of CrySL with domain experts, this design was received positively. We next explain the individual elements that a typical CrySL rule comprises by means of Figure 5.1, which shows an abbreviated CrySL rule for `javax.crypto.Cipher` for which we discussed an extensive example in Section 1.1.

5.1.2 Sections in a CrySL Rule

To provide simple and reusable constructs, a CrySL rule is defined on the level of individual classes. Therefore, the rule starts off by stating the class or interface that it seeks to specify.

In Figure 5.1, the **OBJECTS** section defines several objects[2] to be used in later sections of the rule (e.g., the object `transformation` of type `String`). These objects are typically used as parameters or return values in the **EVENTS** section.

The **EVENTS** section defines all methods that may contribute to the successful use of a `Cipher` object, including two *method event patterns* (Lines 37–38). The first pattern matches calls to `getInstance(String algorithm)`, but the second pattern actually matches calls to two overloaded `getInstance()` methods:

- `getInstance(String algorithm, Provider provider)`
- `getInstance(String algorithm, String provider)`

The first parameter of all three methods is a `String` object whose value states the transformation used for encryption. This parameter is represented by the previously defined `transfomation` object. Two of the `getInstance()` methods are overloaded with two parameters. Since there is no need to specify the second parameter in either method, CrySL allows one to substitute it with an underscore that serves as a placeholder in one combined pattern definition (Line 38). This concept of method event patterns is similar to pointcuts in aspect-oriented programming languages such as AspectJ [KHH+01]. For CrySL, we resort to a more lightweight and restricted syntax as we found full-fledged pointcuts to be unnecessarily complex. Subsequently, the rule defines patterns for the various `init()` methods that set the proper parameter values (e.g., keysize), `update()` methods that encrypt parts of the plaintext and `doFinal()` methods that complete the encryption and return the ciphertext.

Line 53 defines a usage pattern for `Cipher` using the keyword **ORDER**. The usage pattern is a regular expression of method event patterns that are defined in **EVENTS**. Although each method pattern defines a label to simplify referencing related events (e.g., `g1` or `i1`), it is tedious and error-prone to require listing all those labels again in the **ORDER** section. Therefore, CrySL allows defining *aggregates*. An aggregate represents a disjunction of multiple patterns by means of their labels. Line 39 defines an aggregate `Instances` that groups the two `getInstance` patterns. Using aggregates, the usage pattern for `Cipher` reads: there must be exactly one call to one of the `getInstance()` methods, which must be followed by a call to an `init()` method, which may optionally be followed by one or several calls to `update()`. The encryption is completed through a call to `doFinal()`. Methods `update()` to `doFinal()` may be called multiple times.

Following the keyword **CONSTRAINTS**, Lines 57–59 define the constraints for objects listed under **OBJECTS** and used as parameters or return values in the **EVENTS** section. In the abbreviated CrySL rule in Figure 5.1, the first such parameter constraint limits the encryption algorithm of `transformation` to `"AES"` or `"RSA"`. To this end, the rule employs one of CrySL's built-in function: `alg`. To obtain the required expressiveness, we have enriched CrySL with some simple built-in auxiliary functions. The function `alg` extracts the encryption algorithm from `transformation`. This function is necessary because, as mentioned above, `transformation` actually consists of three parameters: algorithm, mode of operation, and padding scheme. For instance, `alg` would extract `"AES"` from `"AES/GCM"` or from `"AES/CBC/PKCS5Padding"`. Table 5.1 lists all functions.

We have further equipped CrySL with built-in predicates in Table 5.2 that facilitate ensuring auxiliary properties on objects or methods. Note the last two predicates `callTo` and

[2] As the example shows, in CrySL, **OBJECTS** also comprise primitive values.

```
26 SPEC javax.crypto.Cipher
27
28 OBJECTS
29   int encmode;
30   java.security.Key key;
31   java.lang.String transformation;
32   byte[] plainText;
33   byte[] cipherText;
34   ...
35
36 EVENTS
37   g1: getInstance(transformation);
38   g2: getInstance(transformation, _);
39   Instances: g1 | g2;
40
41   i1: init(encmode, key);
42   ...
43   Inits: i1 | ...
44
45   ...
46   Updates: ...
47
48   f1: cipherText = doFinal(plainText);
49   ...
50   Finals: f1 | ...
51
52 ORDER
53   Instances, (Init, (Updates*, Finals)+ )+
54
55
56 CONSTRAINTS
57   alg(transformation) in {"AES", "RSA"};
58   alg(transformation) in {"AES"} => mode(transformation) in {"CBC", "CTR"};
59   encmode in {1,2,3,4} //Numbers stand for constants Cipher.{ENCRYPT_MODE,
         DECRYPT_MODE, WRAP_MODE, UNWRAP_MODE}
60   ...
61
62 REQUIRES
63   generatedKey[key, algorithm];
64
65 ENSURES
66   encrypted[cipherText, plainText];
```

Figure 5.1: CrySL Rule for Using `javax.crypto.Cipher`.

Table 5.1: Helper Functions in CrySL.

Function	**Purpose**
alg(*transformation*) mode(*transformation*) padding(*transformation*)	Extract algorithm/mode/padding from `transformation` parameter of `Cipher.getInstance` call.
length(*object*)	Retrieve length of *object*

Table 5.2: Built-in Predicates in CrySL.

Function	Purpose
instanceof(*object*, *type*)*	*Object* must be of type *type* or one of its subtypes
neverTypeOf(*object*, *type*)	Forbid *object* to be of *type*
notHardCoded(*object*)*	Object must not be hardcoded
callTo(*method*)	Require call to *method*
noCallTo(*method*)	Forbid call to *method*

`noCallTo` may seem redundant to sections **`ORDER`** and **`FORBIDDEN`** because they appear to fulfil the same purpose of requiring or prohibiting certain method calls. However, these two predicates go beyond that because they allow for the specification of conditional forbidden and required methods. Since the original publication of CrySL, we have further enhanced it through two new built-in predicates that are marked with an asterisk in Table 5.2. First, we have added predicate `instanceof` when implementing CogniCrypt$_{GEN}$. We refer to 7.2 for a more detailed discussion of the predicate's purpose. We have, after the fact, also extended CogniCrypt$_{SAST}$ to support the predicate, too. Second, we have implemented support for predicate `notHardCoded`. This predicate enables forbidding the hard-coding of the object that is passed to it.

The **`ENSURES`** section is the final construct in a CrySL rule most rules should implement. It allows CrySL to support rely/guarantee reasoning on how classes should be composed. The section specifies custom and rule-specific predicates to govern interactions between different classes. The **`ENSURES`** section specifies what a class guarantees, presuming that the object is used properly. For example, the `Cipher` CrySL rule in Figure 5.1 ends with the definition of a *predicate* `encrypted` with the `cipherText` object and its corresponding plaintext as parameters. This predicate may be *required* (i.e., relied on) by another rule through the keyword **`REQUIRES`** that requires some data to be encrypted.

Similarly, this keyword is also used by the CrySL rule for `Cipher` itself. For the encryption a `Cipher` object performs to be secure, the object *requires* the key it takes in method `init()` to have been generated securely and with the correct algorithm. In Line 63, the corresponding CrySL rule thus defines a predicate `generatedKey` under the keyword **`REQUIRES`**.

CrySL rules can contain several other sections that may not appear so often. We showcase them in the following through the CrySL rule for `PBEKeySpec`. In Figure 5.2, the **`FORBIDDEN`** section specifies methods that must *not* be called, because calling them is always insecure. As explained above, `PBEKeySpec` should use a random salt for deriving a cryptographic key from a password. However, the constructor `PBEKeySpec(char[] password)` does not allow for a salt to be passed, and the implementation in the default provider does not generate one. Therefore, this constructor should not be called, and any call to it should be flagged. Consequently, the CrySL rule for `PBEKeySpec` lists it in the **`FORBIDDEN`** section (Line 87). In the case of `PBEKeySpec`, there is an alternative secure constructor (Line 76). For such cases, CrySL allows one to specify an alternative method event pattern using the arrow notation(⇒) shown in Line 87. Depending on the tool support, these alternatives may either be used for constructive error messages and documentation, or automated fix generation. With **`FORBIDDEN`** events, CrySL's language design deviates a bit from its usual white-listing approach. We made this choice deliberately to keep specifications concise. Without explicit **`FORBIDDEN`** events, one would have to simulate their effect by explicitly listing all events defined on a given type except the one that ought to be forbidden. This would significantly increase the size of CrySL specifications.

In general, predicates are generated for a particular usage whenever it does not use any **`FORBIDDEN`** events, its regular **`EVENTS`** follow the usage pattern defined in the **`ORDER`** section, and if

```
67 SPEC javax.crypto.spec.PBEKeySpec
68
69 OBJECTS
70   char[] passwordd;
71   byte[] salt;
72   int iterationCount;
73   int keylength;
74
75 EVENTS
76   create: PBEKeySpec(password, salt, iterationCount, keylength);
77   clear: clearPassword();
78
79 ORDER
80   create, clear
81
82 CONSTRAINTS
83   iterationCount >= 10000;
84   neverTypeOf(password, java.lang.String);
85
86 FORBIDDEN
87   PBEKeySpec(char[]) => create;
88 ...
89
90 REQUIRES
91         randomized[salt];
92
93 ENSURES
94   speccedKey[this, keylength] after create;
95
96 NEGATES
97   speccedKey[this, _];
```

Figure 5.2: Abbreviated CrySL rule for `javax.crypto.spec.PBEKeySpec`.

```
METHOD :=
    methname(PARAMETERS)

PARAMETERS :=
    varname , PARAMETERS
    varname

TYPES :=
    QualifiedClassName , TYPES
    TYPE

CONSTANTLIST :=
    constant , CONSTANTLIST
    constant

AGGREGATE :=
    label | AGGREGATE
    label ;

EVENT :=
    AGGREGATE
    label : METHOD
    label : varname = METHOD                    A: B = C(D) — a single event with
                                                label A consisting of method C, its
                                                parameter D, and return object B

PREDICATE :=
    predname(PARAMETERS)
    predname(PARAMETERS) after EVENT

PREDICATES :=
    PREDICATE ; PREDICATES
```

Figure 5.3: Basic CrySL Syntax Elements.

the usage fulfils all constraints in the `CONSTRAINTS` section of its corresponding rule. `PBEKeySpec`, however, deviates from that standard. The class contains a constructor that receives a user-provided password, but the method `clearPassword` deletes that password later, making it no longer accessible to other objects that might use the key-spec. Consequently, a `PBEKeySpec` object fulfils its role after calling the constructor but only until `clearPassword()` is called.

To model this usage precisely, CrySL allows one to specify a method-event pattern using the keyword `after` (Line 94). Usually, a predicate is supposed to be generated, when an object of the given type has successfully and fully followed the call pattern given in its `ORDER` section. However, with the `after` keyword, a predicate is generated right after the respective method is called. Furthermore, CrySL supports invalidating an existing predicate in the `NEGATES` section (Line 97). The last call to be made on a `PBEKeySpec` object is the call to `clearPassword()` (Line 80). Additionally, the rule lists the predicate `keySpec[this,_]` within the `NEGATES` block. Semantically, negating a predicate means the following: a final event in the `ORDER` pattern, in this case a call to `clearPassword()`, invalidates the previously generated `keyspec` predicate(s) for `this`. Only existing predicates can be invalidated. Negating a predicate that does not exist (yet or anymore) is not possible. Section 5.2.2 presents the formal semantics of predicates.

For reference, we provide the basic syntactic elements of CrySL and the full syntax in Figures 5.3 and 5.4, respectively.

5.2 CrySL Formal Semantics

CrySL may serve as a basis for multple kinds of tool support. In this section, we therefore present a formal semantics of the language that is tool-independent.

SPEC TYPE;

OBJECTS
OBJECTS :=
OBJECT **;** OBJECTS — *A ; B — a list of objects A and B*
OBJECT **;** — *A — a list of the single object A*
OBJECT :=
TYPE varname — *A B — object B of Java type A*

EVENTS
EVENTS :=
EVENT **;** EVENTS — *A ; B — a list of events A and B*
EVENT **;** — *A — a list of the single event A*

FORBIDDEN
FMETHODS :=
FMETHOD **;** FMETHODS — *A ; B — a list of forbidden A and B*
FMETHOD **;** — *A — a list of the single forbidden method A*
FMETHOD :=
methname(TYPES) => label — *A(B) => C — a forbidden method named A with parameter of Type B and replacement C*

ORDER
USAGEPATTERN :=
USAGEPATTERN **,** USAGEPATTERN — *A , B — A followed by B*
USAGEPATTERN | USAGEPATTERN — *A | B — A or B*
USAGEPATTERN **?** — *A? — A is optional*
USAGEPATTERN ***** — *A* — 0 or more As*
USAGEPATTERN + — *A+ — 1 or more As*
(USAGEPATTERN **)** — *(A) — grouping*
AGGREGATE

CONSTRAINTS
CONSTRAINTS :=
CONSTRAINT **;** CONSTRAINTS
CONSTRAINT => CONSTRAINT — *A => B — A implies B*
CONSTRAINT
CONSTRAINT :=
varname **in {** CONSTANTLIST **}** — *A in {1, 2} — A should be 1 or 2*

REQUIRES
REQ_PREDICATES :=
PREDICATES

ENSURES
ENS_PREDICATES :=
PREDICATES

NEGATES
NEG_PREDICATES :=
PREDICATES

Figure 5.4: CrySL Rule Syntax in Extended Backus-Naur Form (EBNF) [BBG+63].

5.2.1 Basic Definitions

A CrySL rule consists of several sections. The `OBJECTS` section comprises a set of typed variable declarations $\mathbb{V}$. In the syntax in Figure 5.4, each declaration $v \in \mathbb{V}$ is represented by the syntax element `TYPE varname`. $\mathbb{M}$ is the set of all resolved method signatures, where each signature includes the method name and argument types. The `EVENTS` section contains elements of the form (m, v), where $m \in \mathbb{M}$ and $v \in \mathbb{V}^*$. We denote the set of all methods referenced in `EVENTS` by M. The `FORBIDDEN` section lists a set of methods from $\mathbb{M}$ denoted by their signatures; forbidden events cannot bind any variables. The `ORDER` section specifies the usage pattern in terms of a regular expression of labels or aggregates that are in M, i.e., over the defined `EVENTS`. We express this regular expression formally by the equivalent non-deterministic finite automaton (Q, M, δ, q_0, F) over the alphabet M, where Q is a set of states, q_0 is its initial state, F is the set of accepting states, and $\delta : Q \times M \to \mathcal{P}(Q)$ is the state transition function.

The `CONSTRAINTS` section is a subset of $\mathbb{C} := (\mathbb{V} \to \mathcal{O} \cup \mathcal{V}) \to \mathbb{B}$ (i.e., each constraint is a boolean function), where the argument is itself a function that maps variable names in $\mathbb{V}$ to objects in $\mathcal{O}$ or values with primitive types in $\mathcal{V}$.

A CrySL rule is a tuple $(T, \mathcal{F}, \mathcal{A}, \mathcal{C})$, where T is the reference type specified by the `SPEC` keyword, $\mathcal{F} \subseteq \mathbb{M}$ is the set of forbidden events, $\mathcal{A} = (Q, M, \delta, q_0, F) \in \mathbb{A}$ is the automaton induced by the regular expression of the `ORDER` section, and $\mathcal{C} \subseteq \mathbb{C}$ is the set of `CONSTRAINTS` that the rule lists. We refer to the set of all CrySL rules as $\mathbb{SPEC}$.

Our formal definition of a CrySL rule does not contain the sections `REQUIRES`, `ENSURES`, and `NEGATES`. Those sections reason about the interaction of predicates, whose formal treatment we discuss in Section 5.2.2.

5.2.2 Runtime Semantics

Each CrySL rule encodes usage constraints to be validated for all runtime objects of the reference type T stated in its `SPEC` section. We define the semantics of a CrySL rule in terms of an evaluation over a runtime program trace that records all relevant runtime objects and values, as well as all events specified within the rule.

Definition 1 (Event)**.** *Let $\mathcal{O}$ be the set of all runtime objects and $\mathcal{V}$ the set of all primitive-typed runtime values. An* event *is a tuple $(m, e) \in \mathbb{E}$ of a method signature $m \in \mathbb{M}$ and an* environment *e (i.e., a mapping $\mathbb{V} \to \mathcal{O} \cup \mathcal{V}$ of the parameter variable names to concrete runtime objects and values). If the environment e holds a concrete object for the `this` value, then it is called the event's* base object*.*

Definition 2 (Runtime Trace)**.** *A* runtime trace *$\tau \in \mathbb{E}^*$ is a finite sequence of events $\tau_0 \dots \tau_n$.*

Definition 3 (Object Trace)**.** *For any $\tau \in \mathbb{E}^*$, a subsequence $\tau_{i_1} ... \tau_{i_n}$ is called an* object trace *if $i_1 < ... < i_n$ and all base objects of τ_{i_j} are identical.*

Lines 13–14 in Figure 1.1 result in an object trace that has two events:

$$(m_0, \{\texttt{algorithm} \mapsto \texttt{"AES"}, \texttt{this} \mapsto o_{ciph}\})$$
$$(m_1, \{\texttt{algorithm} \mapsto \texttt{"AES"}, \texttt{encmode} \mapsto \texttt{1},$$
$$\texttt{key} \mapsto \texttt{cipherKey}, \texttt{this} \mapsto o_{ciph}\})$$

where m_0 and m_1 are the signatures of the `getInstance()` and `init()` methods of the `Cipher` class. For static factory methods such as `getInstance()`, we assume that `this` is bound to the returned object. We use o_{ciph} to denote that the object `o` is bound to the variable `ciph` at runtime.

$$
\begin{aligned}
sat^o \colon \mathbb{E}^* \times \mathbb{SPEC} &\to \mathbb{B} \\
[\tau^o, (T^o, \mathcal{F}^o, \mathcal{A}^o, \mathcal{C}^o)] &\to sat^o_F(\tau^o, \mathcal{F}^o) \ \wedge \\
&\quad\ sat^o_{\mathbb{A}}(\tau^o, \mathcal{A}^o) \ \wedge \\
&\quad\ sat^o_{\mathbb{C}}(\tau^o, \mathcal{C}^o)
\end{aligned}
$$

Figure 5.5: Function sat^o Verifies an Individual Object Trace for Object o.

The decision whether a runtime trace τ satisfies a set of CrySL rules involves two steps. In the first step, individual object traces are evaluated independently of one another. Yet, different runtime objects may still interact with each other. CrySL rules capture this interaction by means of rely/guarantee reasoning, implemented through predicates that a rule ensures on a runtime object. These interactions between different objects are checked against the specification in a second step by considering the predicates they require and ensure. We first discuss individual object traces in more detail.

Individual Object Traces

The sections `FORBIDDEN`, `ORDER`, and `CONSTRAINTS` are evaluated on individual object traces. Figure 5.5 defines the function sat^o that is true if and only if a given trace τ^o for a runtime object o satisfies its CrySL rule. This definition of sat^o ignores interactions with other object traces. We will discuss later how such interactions are resolved. In the following, we assume the trace $\tau^o = \tau^o_0, ..., \tau^o_n$, where $\tau^o_i = (m^o_i, e^o_i)$. To illustrate the computation, we will also refer to method `encrypt()` from our example in Figure 1.1 and the involved CrySL rule for `Cipher` in Figure 5.1. The function sat^o is composed of three sub-functions:

Forbidden Events (sat^o_F) Given a trace τ^o and a set of forbidden events $\mathcal{F}$, sat^o ensures that none of the trace events is forbidden.

$$sat^o_F(\tau^o, \mathcal{F}^o) := \bigwedge_{i=0...n} m^o_i \notin \mathcal{F}^o$$

The CrySL rule for `Cipher` does not list any forbidden methods. Hence, sat^o trivially evaluates to true for object `ciph` in Figure 1.1.

Order Errors ($sat^o_{\mathbb{A}}$) The second function checks that the trace object is used in compliance with the specified usage pattern (i.e., all methods in the rule are invoked in no other than the specified order). Formally, the sequence of method signatures of the object trace $m^o := m^o_0, \ldots, m^o_n$ (i.e., the projection onto the method signatures) must be an element of the language $\mathcal{L}(\mathcal{A}^o)$ that the automaton $\mathcal{A}^o = (Q, \mathbb{M}, \delta, q_0, F)$ of the `ORDER` section induces. Therefore, it is

$$sat^o_{\mathbb{A}}(\tau^o, \mathcal{A}^o) := m^o \in \mathcal{L}(\mathcal{A}^o).$$

By definition of language containment, after the last observed signature of the trace m^o_n, the corresponding state of the automaton must be an accepting state $s \in F$. This definition ignores any variable bindings. They are evaluated in the second step.

Figure 5.6 displays the automaton created for `Cipher` using the aggregate names as labels. State *0* is the initial state, and state *4* is the only accepting state. Following the code in Figure 1.1 for the object `ciph` of type `Cipher`, the automaton transitions from state *0* to *1*

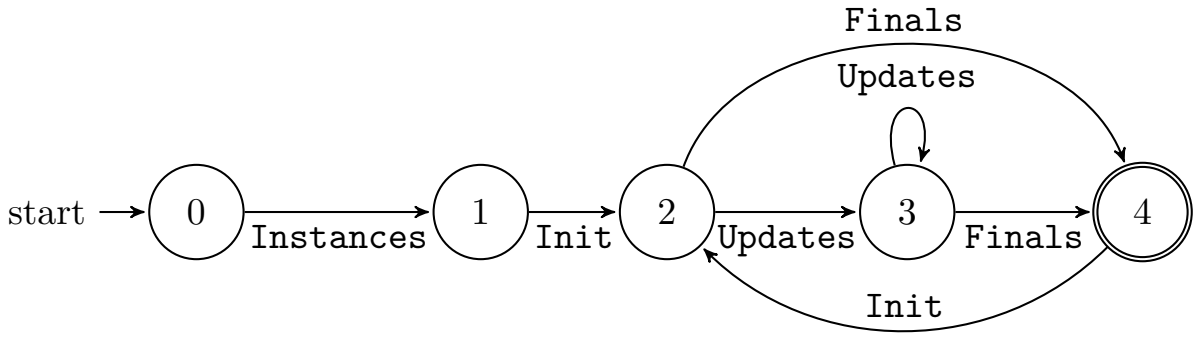

Figure 5.6: State Machine for the CrySL Rule in Figure 5.1 (Without the Implicit Error State).

at the call to `getInstance()` (Line 13). With the calls to `init()` (Line 14) and `doFinal()` (Line 15), the automaton first moves to state *2* and finally to state *4*. Therefore, function $sat^o_{\mathbb{A}}$ evaluates to true for this example.

Constraints ($sat^o_{\mathbb{C}}$) The validity check of constraints ensures that all parameter constraints of $\mathcal{C}$ are satisfied. This check requires the sequence of environments $(e^o_0, ..., e^o_n)$ of the trace τ^o. All objects that are bound to the variables along the trace must satisfy the constraints of the rule.

$$sat^o_{\mathbb{C}}(\tau^o, \mathcal{C}^o) := \bigwedge_{c \in \mathcal{C}^o, i=0...n} c(e^o_i)$$

To compute $sat^o_{\mathbb{C}}$ for the `Cipher` object `ciph` at the call to `getInstance()` in Line 13, only the first two parameter constraints have to be checked. This is because the corresponding environment e^o_{13} holds a value only for `transformation`, and the third constraint references the variable `encmode`. The evaluation function c returns true if the algorithm of `transformation` assumes either `"AES"` or `"RSA"` as its value, which is the case in Figure 1.1. However, the second constraint (Line 58) does not evluate to true. As the algorithm of `transformation` in the example is `"AES"`, the mode of operation must be either `"CBC"` or `"CTR"`. Since the mode is not specified explicitly in the code, `"ECB"` is selected automatically. Therefore, this parameter constraint is not satisfied and object `ciph` should be considered insecure. To gather all violations of the CrySL rule, we do not abort here, but carry on with the evaluation of the example. The computation of $sat^o_{\mathbb{C}}$ for Lines 14–15 works similarly.

Interaction of Object Traces

To define interactions between individual object traces, the **`REQUIRES`**, **`ENSURES`**, and **`NEGATES`** sections allow individual CrySL rules to reference one another. For a rule for one object to hold at any given point in an execution trace, all predicates that its **`REQUIRES`** section lists must have been both previously *ensured* (by other specifications) and not *negated*. Predicates are *ensured* (i.e., generated) and *negated* (i.e., killed) by certain events. Formally, a predicate is an element of $\mathbb{P} := \{(name, args) \mid args \in \mathbb{V}^*\}$ (i.e., a pair of a predicate name and a sequence of variable names). Predicates are generated in specific states. Each CrySL rule induces a function $\mathcal{G}\colon S \to \mathcal{P}(\mathbb{P})$ that maps each state of its automaton to the predicate(s) the state generates.

The predicates listed in the **`ENSURES`** and **`NEGATES`** sections may be followed by the term **`after`** n, where n is a method event pattern label or aggregate. The states that follow the event or aggregate n in the automaton generate the respective predicate. If the term **`after`** is not used

for a predicate, the accepting states of the automaton generate (or negate) that predicate (i.e., we interpret it as **after** n, where n is an event or aggregate that leads to a accepting state).

In addition to states selected as predicate-generating, the predicate is also ensured if the object resides in any state that transitively follows the selected state, unless the states are explicitly (de-)selected for the same predicate within the **NEGATES** section. At any state that generates a predicate, the event driving the automaton into this state binds the variable names to the values that the specification previously collected along its object trace.

Formally, an event $n^o = (m^o, e^o) \in \mathbb{E}$ of a rule r and for an object o ensures a predicate $p = (predName, args) \in \mathbb{P}$ on the objects $e^o \in \mathcal{O}$ if:

1. The method m^o of the event leads to a state s of the automaton that generates the predicate p (i.e., $p \in \mathcal{G}(s)$).
2. The runtime trace of the event's base object o satisfies the function sat^o.
3. All relevant **REQUIRES** predicates of the rule are satisfied at execution of event n^o.

For the `Cipher` object `ciph` in Figure 1.1, a predicate is *not* generated at Line 14. As discussed in this section so far, two object-internal requirements are fulfilled in that (1) its automaton transitions to its only predicate-generating state (state *4* of the automaton in Figure 5.6) and (2) sat^o_F evaluates to true for lack of any forbidden methods that could be called. However, first, as discussed above, $sat^o_{\mathbb{C}}$ evaluates to false because ECB mode is used and the second constraint in the CrySL rule for `Cipher` prohibits this mode.

On top of this constraint violation, one of the *required* predicates is missing. As we described in Chapter 1, class `PBEKeySpec` is misused in several different ways in method `generateKey()` in the example in Figure 1.1. Hence, the object `spec` does not receive a predicate. This lack of a predicate for `spec` is propagated to all objects `spec` flows to, even transitively. Consequently, the `SecretKeySpec` object at the end of method `generateKey()` does not ensure its predicate `generatedKey` either. When this object now flows into the call to `init()` of object `ciph`, it can also not generate a predicate.

In summary, no predicate is generated for `ciph`, because it violates one parameter constraints and the object `cipherKey` does not hold a `generatedKey` predicate.

5.3 Implementation

We have implemented a CrySL compiler on top of Xtext [Xte20], an open-source framework for developing domain-specific languages. Given the CrySL grammar, Xtext provides a parser, type checker, and syntax highlighter for the language. When supplied with a type-safe CrySL rule, Xtext outputs the corresponding AST. We have further developed a parser that translates CrySL rules into an object model that may be consumed by any tool support building on top of CrySL, and is indeed used by CogniCrypt$_{\text{SAST}}$ and CogniCrypt$_{\text{GEN}}$.

Rule Set for the JCA

We have developed the most comprehensive set of usage rules for Crypto APIs in Java to date. Our rule set compasses rules of the JCA, the JSSE, BouncyCastle, BouncyCastle as a JCA provider, and Google Tink. In the following, we discuss the rule set for the JCA, which, for the remainder of the thesis, we will refer to as Ruleset$_{\text{FULL}}$, in more detail. It comprises all relevant classes and interfaces. In an iterative specification process, we first worked through the JCA documentation to produce a set of rules and then refined these rules through selective

discussions with cryptographers and searching security blogs and forums. In total, we have devised 23 rules. Apart from the rules we have discussed for `PBEKeySpec` and `Cipher`, the full rule set encompasses CrySL specifications that specify correct uses of all JCA classes, which offer various cryptographic services. In the following, we describe these services with their respective classes and briefly summarize important usage constraints. All mentioned classes are defined in the JCA packages `javax.crypto` and `java.security`.

Asymmetric Key Generation Asymmetric and symmetric cryptography requires different key formats. Asymmetric cryptography uses pairs of public and private keys. While one of the keys encrypts plaintexts to ciphertexts, the second key decrypts the ciphertext. The JCA models a key pair as class `KeyPair` whose instances are generated by `KeyPairGenerator`.

Symmetric Key Generation Symmetric cryptography uses the same key for encryption and decryption. The JCA models symmetric keys as type `SecretKey`, generated by a `SecretKeyFactory` or `KeyGenerator`. The `SecretKeyFactory` also enables password-based cryptography using `PBEParameterSpec` or `PBEKeySpec`.

Signing and Verification of Data The class `Signature` of the JCA allows one to digitally sign data and verify a signature based on a private/public key pair. A `Signature` requires the key pair to be correctly generated, hence the rule for `Signature` **`REQUIRES`** a predicate from the asymmetric-key generation task.

Generation of Initialization Vectors Initialization vectors (IVs) are used to add entropy to ciphertexts of encryptions. An IV must have enough randomness and must be properly generated. The JCA class `IvParameterSpec` wraps a byte array as an IV and it is required for the array to be randomized by `SecureRandom`. The CrySL rule for `IvParameterSpec` **`REQUIRES`** a predicate `randomized`.

Encryption and Decryption The key component of the JCA is represented by the class `Cipher`, which implements functionality to encrypt or decrypt data. Depending on the used algorithms, modes and paddings must be selected and keys and initialization vectors must be properly generated. Hence, the complete CrySL rule for `Cipher` requires many other cryptographic services to be executed securely earlier and list them in its respective **`REQUIRES`** clause.

Hashing & MACs There are two forms of cryptographic hash functions. A MAC is an authenticated hash that requires a symmetric key, but there are also keyless hash functions such as MD5 or SHA-256. The JCA's class `Mac` implements functionality for mac-ing, while keyless hashes are computed by `MessageDigest`.

Persisting Keys Securely storing key material is an important cryptographic task for confidentiality and integrity of the encrypted data. The JCA class `KeyStore` supports developers in this task and stores the key material.

Cryptographically Secure Random-Number Generation Randomness is vital in all aspects of cryptography. Java offers cryptographically secure pseudo-random number generators through `SecureRandom`. As discussed for `PBEKeySpec`, `SecureRandom` often acts as a helper and therefore many rules list the `randomized` predicate in their own **`REQUIRES`** section.

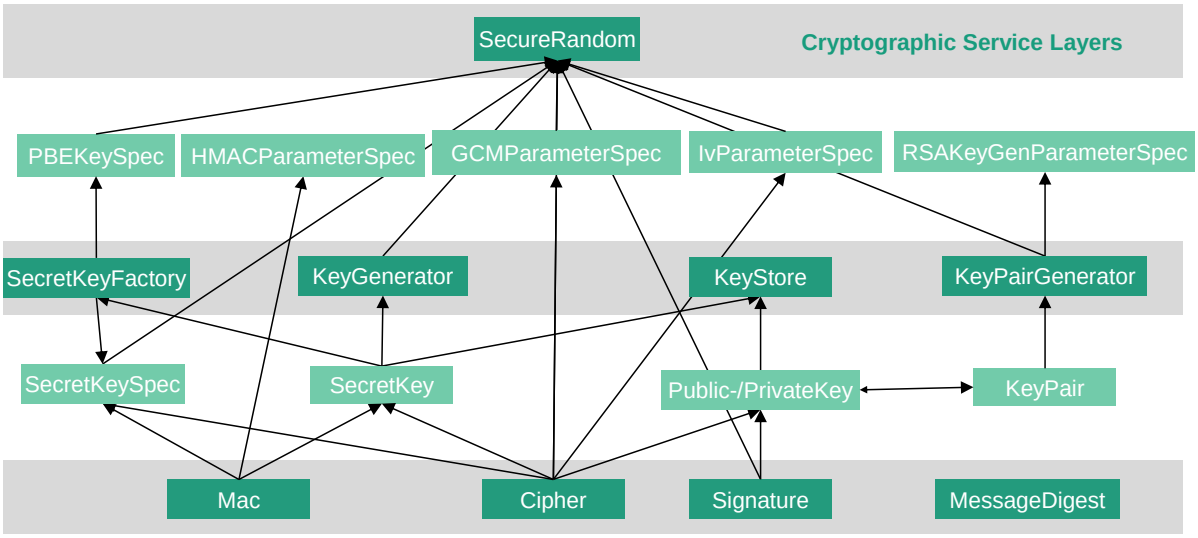

Figure 5.7: Dependency Layers in JCA.

Combination of Different Cryptographic Services In practice, cryptographic services are often combined to achieve more security goals than one primitive could offer on its own. One often-used example is *authenticated encryption* that achieves not only confidentiality, but also authenticity and integrity on the original plaintext. To this end, MACs and encryption are combined. While there are multiple ways to combine the two, only first encrypting the plaintext and then computing the MAC on the ciphertext is recommended ("*encrypt-then-mac*") [fISB17]. As such combinations of different cryptographic services are implemented through source code as well, we have explicitly encoded secure combinations in the rules of participating classes through predicates.

Dependencies between Rules When developing the rule set, we noticed dependencies between different rules that are expressed through predicates forming the structure in Figure 5.7. That is, there are five distinct layers of classes with relations not within any given layer but only across them. The three layers on grey background provide cryptographic services, the other two comprise data holders. At the top, there is `SecureRandom` that supplies randomization services for all other layers. The second layer from the top involves classes for parameters of cryptographic operations (e.g., key specifications, initialization vectors). They do not provide any cryptographic operations themselves, but rather represent helper objects that are required for such functionality. The third layer exclusively revolves around key generation and management, with the fourth layer containing the keys generated by the third layer. The bottom layer contains all classes that provide cryptographic functionality to end-users (e.g., encryption, signing, hashing).

Structural Overview All 23 rules define a usage pattern. Some classes (e.g. `IvParameterSpec`) contain one call to a constructor only, while others (e.g., `Cipher`) involve almost ten elements with several layers of nesting. Fifteen rules come with parameter constraints, eight of which contain limitations on cryptographic algorithms. The eight rules without parameter constraints

are mostly related to classes whose purpose is to set up parameters for specific encryptions (e.g., `GCMParameterSpec`). All rules define at least one **`ENSURES`** predicate, while only eleven require predicates from other rules. Across all rules, we have only declared two methods forbidden. We do not find this low number surprising as such methods are always insecure and should not at all be part of any security API. If at all, two forbidden methods is too high a number. All rules are available at `https://github.com/CROSSINGTUD/Crypto-API-Rules`.

5.4 Limitations

CrySL comes with certain limitations in terms of expressiveness. By design, the usage constraints defined by a CrySL rule are tied to the API and how it may be used. More high-level properties that are not directly enforceable through the API under specification may therefore not be expressible through CrySL. That may in particular relate to temporal properties such as regular re-keying or prohibiting the re-using of nonces. For APIs such as the JCA that provide no means to ensure such properties, the corresponding CrySL rules may not express them either.

Further imprecision may result from CrySL's white-listing approach. Specifications, especially when they are manually crafted, are almost necessarily incomplete. In the case of black-listing specifications, that usually means that incorrect uses of an API are accepted as correct. White-listing approaches, on the other hand, rather prohibit uses that should be considered correct. We minimized this problem by designing CrySL such that methods that are not defined in the **`EVENTS`** section have no impact on whether or not an object is used correctly. This restriction also saves specification effort, the overall problem still persists, however.

The issue is slightly aggravated by CrySL's binary notion of security: An algorithm, a keysize, a chain of method calls is either secure or not. This restriction does not allow for more nuanced distinction that some algorithms may not be entirely insecure, but may only be used under circumstances (e.g., ECB when only one block of plaintext is encrypted [fISB17] or CBC/PKCS5Padding when attackers cannot gather information about whether or not a ciphertext they have is correctly padded (i.e., in the absence of a padding oracle) [fISB17]). It further ignores the application context in which algorithms may be used. The hashing algorithm MD5 should no longer be used for security purposes, but can be — and often is — used for file hashing. In each case, we had to decide whether we would allow a use or not.

In the context of cryptography, we view this limitation as an acceptable trade-off. Cryptography is likely being used in sensitive circumstances. Being incorrectly warned about a correct usage, to us, appears preferable to missing one and, by extension, making an application insecure. That said, when devising CrySL rules, we have carefully read the official API documentation to minimize such imprecision. For other domains, mileage may also vary. We leave it to future work to measure the effects overly restrictive rules have in practice.

CrySL further suffers from lacking any awareness of the environment an API is used in. This lack of awareness becomes an issue when, like with `SecureRandom`, the selection of algorithms that are available for a cryptographic operation depends on the operating system. Windows machines have other PRNG algorithms available than Unix machines [Ora19c]. In certain Java versions, class `CipherInputStream` exhibits a vulnerability when used in the context of authenticated encryption that causes the encryption to no longer guarantee integrity [Hec14]. Neither example can currently be appropriately expressed in CrySL. Extending CrySL to support environment variables would solve these problems.

The latter example may also be fixed by introducing rules for different Java versions. In the CrySL rules for the corresponding version, the respective method could be marked as **`FORBIDDEN`**. However, *as presented in this thesis*, CrySL does not have a module system. Instead,

one would have to write all rules manually. A module system is in the works and should be presented around Summer 2020.

5.5 Related Work

While we are not aware of any specification languages with CrySL's purpose, other API-specification languages do exist. Some work has also attempted to mine usage rules for Crypto APIs. Our review of these approaches shows that existing specification languages are not optimally suited for defining misuses of Crypto APIs. Similarly, we have also investigated approaches that attempt automated inference of correct uses of Crypto APIs. Based on our findings, we conclude that this goal is hard to achieve.

5.5.1 Languages for Specifying and Checking API Properties

There is a significant body of research on textual specification languages that ensure API properties. Tracematches [AAC+05] were designed to check typestate properties defined by regular expressions over runtime objects. Bodden et al. [Bod10, BLH12] as well as Naeem and Lhoták [NL08] present algorithms to (partially) evaluate state matches prior to program execution, using static analysis.

Martin et al. [MLL05] present Program Query Language (PQL) that enables a developer to specify patterns of event sequences that constitute potentially defective behaviour. A dynamic analysis (i.e., tracematches optimized by a static pre-analysis) matches the patterns against a given program run. A pattern may include a fix that is applied to each match by dynamic instrumentation. PQL has been applied to detecting security-related vulnerabilities such as memory leaks [MLL05], SQL injection, and cross-site scripting [LL05]. Compared to tracematches, PQL captures a greater variety of pattern specifications, at the disadvantage of only flow-insensitive static optimizations. PQL serves as the main inspiration for CrySL's syntax. Other languages that pursue similar goals include PTQL [GOA05], PDL [MVW07], SLIC [BR01, BR02] and TS4J [Bod14].

We investigated tracematches and PQL in detail, yet found them insufficiently equipped for the task at hand. First, both systems follow a black-list approach by defining and finding incorrect program behaviour. We initially followed this approach for crypto-usage mistakes, but quickly discovered that it would lead to long, repetitive, and convoluted misuse-definitions. Consequently, CrySL defines desired behaviour, which, in the case of Crypto APIs, leads to more compact specifications. Second, the above languages are general-purpose languages for bug finding, which causes them to miss features essential to define secure usages of Crypto APIs in particular (e.g., White listing, constraints on String objects and integers). The strong focus of CrySL on cryptography allows us to cover a greater portion of cryptography-related problems in CrySL compared to other languages, while at the same time keeping CrySL relatively simple. Third, thanks to its white-listing approach, CrySL facilitates various kinds of tool support to be built on top of it. The other languages discussed in this section remain on the level of providing bug specifications that program-analysis tools may check for.

5.5.2 Inference/Mining of API-usage Specifications

As an alternative to specifying API-usage properties manually, one can attempt to infer them from existing program code. Robillard et al. [RBK+13] survey over 60 approaches to API property inference. As this survey shows, all but two of the surveyed approaches infer patterns from client code (i.e., from applications that use the API in question). Crypto APIs, however, are misused often, making the raw mining of their uses fairly pointless.

To infer Crypto-API rules, Paletov et al. [PTRV18] thus follow a different approach: instead of mining client code directly, they mine code *changes* related to Crypto APIs. Subsequently, the authors cluster these changes and derive a usage rule from each cluster. While the work is a first step towards inferring Crypto-API rules, it also shows the challenges involved. For instance, a closer observation of the inferred rules reveals that many of them are overly simplistic and lack context. For instance their rule R4 states "SecureRandom with getInstanceStrong() should be avoided" although this is only true "on server-side code running on Solaris/Linux/MacOS"—in most other cases, calling `getInstanceStrong()` is actually recommended and avoids security pitfalls. The approach also lacks recall: the paper states 13 usage rules only, while our rule set for the JCA alone compactly encodes hundreds of individual usage rules. Nonetheless, it would be interesting to see if the authors' approach can be used to infer at least partial CrySL rules. Furthermore, for their experiments, Paletov et al. did not automate the actual generation of machine-checkable rules but instead derived appropriate static checks by hand. This manual step reduces the practicality of their approach.

Gao et al. [GKL+19] follow a similar approach. They formulate the basic assumption underlying their mining approach as "Developers update API usage instances to fix misuses". To this end, they apply CogniCrypt$_{SAST}$ bootstrapped with the JCA rule set on almost 40,000 Android apps. For each of them, they have at least ten different versions, resulting in 600,000 apk files being analyzed. Through tracking code *changes* over multiple versions, they aim to mine fixes and, by extension, correct uses of Crypto APIs. Unfortunately, their results show that only about half of code *updates* addressing Crypto APIs fix a previously introduced misuse. Their basic assumption is therefore proven wrong and no rules can be mined.

Another idea that appears sensible at first sight is to infer rules from posts on developer portal. Braga and Dahab [BD16] mine misuses from three forums often trafficked by Java developers. They manage to compile quite a comprehensive set of common JCA misuses. Unfortunately, their mining process relies on manual analysis of these posts. We do not expect the amount of work going into developing CrySL rules for an API to be much greater than for this kind of analysis. To yield the full potential of rule inference, the approach should be automated. In contrast, learning *correct* uses is likely even more challenging because studies show the "solutions" posted on these portals often include insecure code [FBX+17, ABF+16, ASW+17]. More recent research by Fischer et al. [FXK+19] that we discussed in depth in Section 4.3 may be promising in overcoming these issues, but, so far, it has not been applied to rule inference.

In result, we conclude that automated mining of API-usage specifications is very challenging for Crypto APIs. We leave to future work to investigate the feasibility of a semi-automated approach in which at least partial CrySL rules are inferred automatically, to then be refined and completed by security experts.

5.6 Conclusion

In this chapter, we have proposed API-usage-specification language CrySL. Each CrySL rule is specific to one class, and it may include usage pattern definitions and constraints on parameters. Predicates model the interactions between classes. For example, a rule may generate a predicate on an object if it is used successfully, and another rule may require that predicate from an object it uses.

CrySL is the first step to systematically addressing cryptographic misuse. CrySL enables cryptography experts to specify how to correctly use a Crypto API. By means of CrySL, we thus extract usage specifications into their own language. This flexibility facilitates several things. First, CrySL comes with its own tool-independent semantics. Hence, in contrast to other specification languages, it is not tied to one particular tool support for Crypto APIs. Instead,

specifications in CrySL may serve as building blocks for multiple kinds of tool support, including but not limited to program analysis, code generator or program repair. To facilitate this flexible use further, we also provide an extra parser that translates CrySL rules into a CrySL object model more suitable for Java-based tool support. Second, updating rules is not left to the expert of the given tool support (e.g., the static analyzer) because they are not hard-coded into the tool. Instead, crypto experts may make theses changes themselves. Third, when this aspect leads to greater security when some cryptographic algorithm becomes broken or some API is found to implement digital signatures insecurely. Cryptography experts are likely to be more aware of the latest cryptography research than the software engineers who have built one particular checking tool that may have been abandoned a few years ago.

We see several avenues for future work. First, one should address the limitations we have discussed. CrySL would particularly benefit from support for a more nuanced and contextual understanding as well as environment variables. In our work, we have applied CrySL to cryptographic APIs in Java. Yet, the concepts of the approach are not specific to Java. Exploring the application of CrySL to other languages may therefore prove an interesting venture. Similarly, there are other kinds of APIs, including other security APIs, that might benefit from CrySL specifications. Future work should also model more APIs and potentially extend the language to suit these APIs' needs. Lastly, a wide variety of CrySL-based tool support is possible. In the next two chapters, we show only two options out of many.

6

CogniCrypt$_{\text{sast}}$

In this chapter, we present our first CrySL-based tool support: CogniCrypt$_{\text{sast}}$. It uses the CrySL compiler that parses and type-checks CrySL rules and translates them into the CrySL object model. CogniCrypt$_{\text{sast}}$ then transforms the rules in the CrySL object model into an efficient, yet precise flow-sensitive and context-sensitive static data-flow analysis. The analysis automatically checks a given Java or Android app for compliance with the encoded CrySL rules.

To evaluate CogniCrypt$_{\text{sast}}$, we use it—bootstrapped with our JCA rule set—to conduct three studies, two of which on large application corpora. First, we scan 10,000 Android apps. We have also modelled the existing hard-coded rules by Egele et al. [EBFK13] in CrySL and compared the findings of the generated static analysis to those of CogniCrypt$_{\text{sast}}$ for the 10,000 Android apps. Our more comprehensive rule set reports three times more violations, most of which are true warnings. With such comprehensive rules, CogniCrypt$_{\text{sast}}$ finds at least one misuse in 95% of the apps. CogniCrypt$_{\text{sast}}$ is also highly efficient: for more than 75% of the apps, the analysis finishes in under three minutes per app, where most of the time is spent in Android-specific call graph construction.

In the second study, we run CogniCrypt$_{\text{sast}}$ on 250 security-critical apps. With this study, we aim to investigate a more specific domain of Android apps, of which we would expect significantly less misuse. Our findings reveal the number of apps containing misuses indeed to be lower, with 71% of apps misusing Crypto APIs at least once. However, 71% is still too large a number for this domain.

Lastly, we apply CogniCrypt$_{\text{sast}}$ to *all* 204,788 software artefacts on Maven Central, the world's largest Java code repository, and present the first comprehensive study of misuses of crypto APIs in Java. This study facilitates an investigation into whether there is a difference between average developers for Java and Android in terms of how securely they use cryptographic APIs. We find this matter worthy of investigation as we would assume regular Java code to contain significantly fewer misuses due to the relative maturity of Java as a language and breadth of application fields. Across all analyzed artefacts, CogniCrypt$_{\text{sast}}$ finds 24,349 cryptography misuses in 5,712 Java artefacts. More than 63% of all artefacts that use the JCA contain at least one misuse. We, therefore, conclude that Java code is indeed less insecure, but overall still not secure.

6.1 Detecting Misuses of Crypto APIs

Please note: Throughout this section, we remain on the level of a workflow description for CogniCrypt$_{\text{sast}}$. More in-depth discussion of the underlying analysis, including a presentation

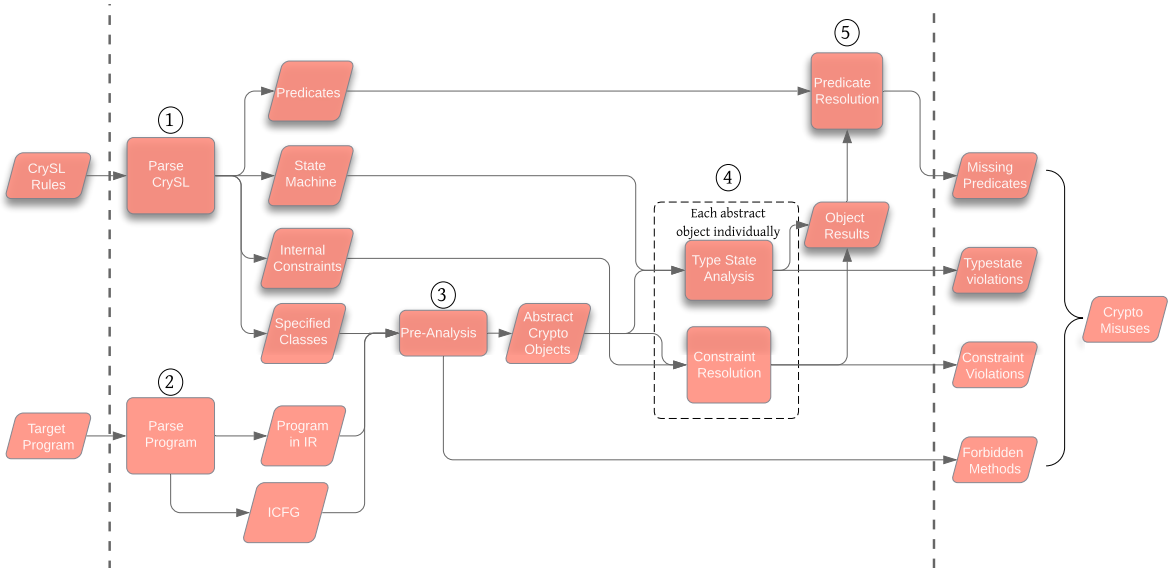

Figure 6.1: The General Workflow of CogniCrypt$_{SAST}$.

```
98  boolean isPrime = isPrime(66); //some non-trivial predicate returning false
99  byte[] input = "Message".getBytes("UTF-8");
100
101 String alg = "SHA-256";
102 if (isPrime) alg = "MD5";
103 MessageDigest md = MessageDigest.getInstance(alg);
104
105 if (input.size() > 0) md.update(input);
106 byte[] digest = md.digest();
```

Figure 6.2: An Example Illustrating the Usage of `java.security.MessageDigest` in Java.

of its flow functions, can be found in the Ph.D. thesis [Spä19] of co-author on both the original ECOOP publication as well as the extended version published at TSE Johannes Späth.

To detect all violations of CrySL rules, CogniCrypt$_{SAST}$ approximates the evaluation function sat^o (Section 5.2.2) using a static data-flow analysis. In the following, we will explain how CogniCrypt$_{SAST}$ does so by means of Figure 6.1. We will also discuss where CogniCrypt$_{SAST}$ deviates from sat^o and why certain approximations are required, using the example in Figure 6.2. We will show later in Section 6.3 that, in practice, our analysis is highly precise and that the chosen approximations rarely actually lead to false warnings.

Let us first briefly discuss the code example in Figure 6.2 that implements a hashing operation. By default, the code uses `SHA-256`. However, if the condition `isPrime` evaluates to true, `MD5` is chosen instead (Line 102). The CrySL rule for `MessageDigest`, displayed in Figure 6.3, does not allow the usage of `MD5` though, because it is no longer secure [fISB17].

The `update()` operation is performed only on non-empty input (Line 105). Otherwise, the call to `update()` is skipped and only the call to `digest()` is executed without any input. A hash function used without any input does not comply with the CrySL rule for `MessageDigest`; it is most likely a programming error as no content is being hashed.

To analyze this piece of code, CogniCrypt$_{SAST}$ parses the CrySL rules made available to it, including the one for `java.security.MessageDigest` and transforms them into several different artefacts ①. First, it assembles a list of the Java classes it has received CrySL rules for. Second, CogniCrypt$_{SAST}$ collects forbidden methods from the rules. Third, for each

```
107 SPEC java.security.MessageDigest
108
109 OBJECTS
110   java.lang.String algorithm;
111   byte[] input;
112   int offset;
113   int length;
114   byte[] hash;
115   ...
116
117 EVENTS
118   g1: getInstance(algorithm);
119   g2: getInstance(algorithm, _);
120   Gets := g1 | g2;
121   ...
122   Updates := ...;
123
124   d1: output = digest();
125   d2: output = digest(input);
126   d3: digest(hash, offset, length);
127   Digests := d1 | d2 | d3;
128
129   r: reset();
130
131 ORDER
132   Gets, ((Updates*, r) |(d2 | (Updates+, Digests)))+
133
134 CONSTRAINTS
135   algorithm in {"SHA-256", "SHA-384", "SHA-512"};
136
137 ENSURES
138   digested[hash, ...];
139   digested[hash, input];
```

Figure 6.3: CrySL Rule for `java.security.MessageDigest`.

rule, CogniCrypt$_{SAST}$ further creates a) a state machine, b) parameter constraints, and c) its predicates. Those artefacts map onto the sections **ORDER**, **CONSTRAINTS**, and **ENSURES** of the corresponding CrySL rules, respectively. As a second step, CogniCrypt$_{SAST}$ parses the program into a program-analysis IR and ICFG ②.

To determine which objects must be analyzed further, i.e., which objects are of types CogniCrypt$_{SAST}$ has a CrySL rule for, CogniCrypt$_{SAST}$ conducts a pre-analysis ③. To this end, CogniCrypt$_{SAST}$ models each runtime object o by its allocation site. In the pre-analysis, CogniCrypt$_{SAST}$ considers all `new` expressions and static calls to `getInstance()` that return an object of a relevant type as relevant allocation sites. The pre-analysis returns a list of *abstract crypto objects (ACO)*. Each ACO represents an allocation site, the calls made on this allocation site in the target program as well as potential runtime values of parameters in these calls. To determine the latter, CogniCrypt$_{SAST}$ iterates through the calls along the ICFG. This step also facilitates the detection of forbidden calls along the way and, thereby, approximate sat^o_F on all ACOs. Depending on the precision of the ICFG, the analysis may find calls to forbidden methods that cannot be reached at runtime. However, since the CrySL rule for `MessageDigest` does not define any forbidden calls, CogniCrypt$_{SAST}$ does not report any for the example regardless.

After the pre-analysis, CogniCrypt$_{SAST}$ moves on to analyzing each ACO individually ④. For each ACO, CogniCrypt$_{SAST}$ first approximates $sat^o_{\mathbb{A}}$ by evaluating the state machine associated with the type of the corresponding allocation site using a typestate analysis. Any deviation from the state machine specified in the corresponding CrySL rule results in CogniCrypt$_{SAST}$ reporting a misuse. However, there is potential for imprecision. CogniCrypt$_{SAST}$'s typestate analysis abstracts runtime traces by program paths. If the typestate analysis is path-insensitive, at branch points it always assumes that both sides of the branch may execute. In our contrived example in Figure 6.2, this feature leads to a false positive: although the condition in Line 105 always evaluates to true, and the call to `update()` is never actually skipped, the analysis considers that this may happen and thus reports a rule violation.

CogniCrypt$_{SAST}$ next approximates $sat^o_{\mathbb{C}}$ by means of a constraint solver. The constraint solver first filters out all *irrelevant* constraints. A constraint is irrelevant if it refers to one or more variables CogniCrypt$_{SAST}$ has not found in the code when constructing the ACOs and has, therefore, not extracted any potential runtime values for. As per Figure 6.3, the rule for `MessageDigest` only includes one parameter constraint—on variable `algorithm`. If we added a new parameter constraint to the rule about the variable `offset`, the constraint solver would filter it out as irrelevant when analyzing the code in Figure 6.2, because the only method this variable is associated with (`digest()` labelled `d3`) is never called. CogniCrypt$_{SAST}$ distinguishes between a variable not being in the source code and not being able to extract values for a variable. With the same rule and code snippet, if the analysis fails to extract the value for `algorithm`, the constraint evaluates to false because CogniCrypt$_{SAST}$ could not prove that it holds. Collecting potential values of a variable over all possible program paths of an allocation site may lead to further imprecision. In our example, CogniCrypt$_{SAST}$ cannot statically rule out that `algorithm` may be `MD5`. The rule forbids the usage of `MD5`. Therefore, the analysis reports a misuse.

Handling predicates in our analysis follows the formal description very closely ⑤. If sat^o evaluates to true for a given allocation site, the analysis checks whether all required predicates for the allocation site have been ensured earlier in the program. In the trivial case, when no predicate is required, CogniCrypt$_{SAST}$ immediately ensures the predicate defined in the **ENSURES** section. CogniCrypt$_{SAST}$ constantly maintains a list of all ensured predicates, including the statements in the program that a given predicate can be ensured for. If the allocation site under analysis requires predicates from other allocation sites, the analysis consults the list of ensured predicates and checks whether the required predicate, with matching names and arguments, exists at the

given statement. If the analysis finds all required predicates, it ensures the predicate(s) specified in the `ENSURES` section of the rule.

6.2 Implementation

CogniCrypt$_{SAST}$ first employs the CrySL parser to parse a set of CrySL rules into the CrySL object model. To conduct the program analysis, it further consists of several extensions to the program analysis framework Soot [VGH+00, LBLH11]. Soot transforms a given Java program into the intermediate representation Jimple, which facilitates executing intra- and inter-procedural static analyses. The framework provides standard static analyses such as call-graph construction. Additionally, Soot can analyze a given Android app intra-procedurally. Further extensions by FlowDroid [ARF+14] enable the construction of Android-specific call graphs necessary to constrcut the ICFG and perform inter-procedural analysis.

Validating the `ORDER` section in a CrySL rule requires solving the typestate check $sat^{o}_{\mathbb{A}}$. To this end, we use IDE^{al}, a framework for efficient inter-procedural data-flow analysis [SAB17], to instantiate a typestate analysis. The analysis defines the finite-state machine $\mathcal{A}^{o}$ to check against, and the allocation sites to start the analysis from. From those allocation sites, IDE^{al} performs a flow-, field-, and context-sensitive typestate analysis.

The constraints and the predicates require knowledge about objects and values associated with rule variables at given execution points in the program. The typestate analysis in CogniCrypt$_{SAST}$ extracts the primitive values and objects on the fly, where the latter are abstracted by allocation sites. When the typestate analysis encounters a call site that is referred to in an event definition, and the respective rule requires the object or value of an argument to the call, CogniCrypt$_{SAST}$ triggers an on-the-fly backward analysis to extract the objects or values that may participate in the call. This on-the-fly analysis yields comparatively high performance and scalability, because many of the arguments of interest are values of type `String` and `int`. Thus, using an on-demand computation avoids having to propagate *all* strings and integers through the program. For the on-the-fly backward analysis, CogniCrypt$_{SAST}$ uses an extended version of the on-demand pointer analysis Boomerang [SDAB16b] to propagate both allocation sites and primitive values. Once the typestate analysis is completed, and all required queries to Boomerang are computed, CogniCrypt$_{SAST}$ solves the parameter constraints and predicates using our own custom-made solvers.

Apart from being integrated into CogniCrypt, CogniCrypt$_{SAST}$ may also be operated as a standalone command-line tool. This way, it takes a program as input and produces an error report detailing misuses and their locations.

6.3 Crypto-API Misuse in Android Apps

We first evaluate CogniCrypt$_{SAST}$ by addressing the following research questions:

RQ1 What are precision and recall of CogniCrypt$_{SAST}$?

RQ2 What types of misuses does CogniCrypt$_{SAST}$ find in Android apps?

RQ3 How fast does CogniCrypt$_{SAST}$ run?

RQ4 How does CogniCrypt$_{SAST}$ compare to the state of the art?

To answer these questions, we apply the static analysis CogniCrypt$_{SAST}$ to 10,000 Android apps from the AndroZoo dataset [ABKT16], using our full CrySL rule set for the JCA

RuleSet$_{\text{full}}$ (Section 5.3). We run our experiments on a Debian virtual machine with sixteen cores and 64 GB RAM. We run CogniCrypt$_{\text{sast}}$ once per application and cap the time of each run to 30 minutes. We choose apps that are available in the official Google Play Store and received an update in 2017. This restriction ensures that we do not report on outdated usages of Crypto APIs. We make available all artefacts at this Github repository: `https://github.com/CROSSINGTUD/paper-crysl-reproduciblity-artefacts`.

6.3.1 Precision and Recall (RQ1)

Setup

To compute precision and recall, we manually check 50 randomly selected apps from our dataset for typestate errors and violations of parameter constraints. To collect this random sample, we implement a Java program that generates random numbers using `SecureRandom` and retrieve the apps from the corresponding lines in the spreadsheet containing the results of analysing the 10,000 apps. We did not check for unsatisfied predicates or forbidden events because these are hard to detect manually—while it may seem simple to check for calls to forbidden events, it is non-trivial to determine whether or not such calls reside in dead code. We compare the results of our manual analysis to those reported by CogniCrypt$_{\text{sast}}$. The goal of this evaluation is to compute precision and recall of the analysis implementation in CogniCrypt$_{\text{sast}}$, not the quality of our CrySL rules. We discuss the latter in Section 6.3.4. Consequently, we define a false positive to be a warning that should not be reported according to the specified rule, irrespective of that rule's semantic correctness. Similarly, a false negative would arise if CogniCrypt$_{\text{sast}}$ missed to report a misuse that, according to the CrySL rule, does exist in the analyzed program.

Results

In the 50 apps we inspected, CogniCrypt$_{\text{sast}}$ detects 228 usages of JCA classes. Table 6.1 lists the misuses that CogniCrypt$_{\text{sast}}$ finds (156 misuses in total). In particular, CogniCrypt$_{\text{sast}}$ issues 27 typestate-related warnings, with only 2 false positives. Both arise because the analysis is path-insensitive (Section 6.2). We further found 4 false negatives that are caused by initializing a `MessageDigest` or a `MAC` object without completing the operation. CogniCrypt$_{\text{sast}}$ fails to find these typestate errors because the supporting off-the-shelf alias analysis Boomerang times out, causing CogniCrypt$_{\text{sast}}$ to abort the typestate analysis without reporting a warning for the object at hand. A larger timeout or future improvements to the alias analysis Boomerang would avoid this problem.

The automated analysis finds 129 constraint violations. We were able to confirm 110 of them. In the other 19 cases, highly obfuscated code causes the analysis to fail to extract possible runtime values statically. For such values, the constraint solver reports the corresponding constraint as violated. A better handling of such highly obfuscated code can be enabled by techniques complementary to ours. For instance, one could augment CogniCrypt$_{\text{sast}}$ with the hybrid analysis Harvester [RAMB16]. We have also checked the apps for missed constraint violations (false negatives), but we were unable to find any.

RQ1: In our manual assessment, the typestate analysis achieves high precision (92.6%) and recall (86.2%). The constraint resolution has a precision of 85.3% and a recall of 100%.

Table 6.1: Correctness of CogniCrypt$_{SAST}$ warnings.

	Total Warnings	**False Positives**	**False Negatives**
Typestate	27	2	4
Constraints	129	19	0
Total	156	21	4

Table 6.2: Types of API Misuses reported by CogniCrypt$_{SAST}$ for Android Apps that use the JCA.

API Misuse Type	**# Warnings**	**# Apps**
Incorrect calling sequences	4,708 (23.0%)	2,896
Incorrect parameter values	11,178 (54.7%)	3,955
Calls to forbidden methods	97 (0.5%)	62
Insecure compositions	4,443 (21.8%)	1,367
Total	20,426	4,143

6.3.2 Types of Misuses (RQ2)

Setup

We report findings obtained by analyzing all our 10,000 Android apps from AndroZoo [ABKT16]. Based on the results of our manual analysis (Section 6.3.1), we discuss and draw hypotheses about our findings on the large scale.

CogniCrypt$_{SAST}$ detects the usage of at least one JCA class in 8,422 apps. Further investigation unveils that many of these usages originate from the same common libraries included in the applications. To avoid counting the same crypto usages twice, and to prevent over-counting, we exclude usages within packages `com.android`, `com.facebook.ads`, `com.google` or `com.unity3d` from the analysis.

Results

Excluding the findings in common libraries, CogniCrypt$_{SAST}$ detects the usage of at least one JCA class in 4,349 apps (43% of the analyzed apps). Most of these apps (95%) contain at least one misuse. We detail CogniCrypt$_{SAST}$'s findings on apps that do contain misuses in Table 6.2. Across all apps, CogniCrypt$_{SAST}$ started its analysis for a total of 40,295 allocation sites (i.e., abstract crypto objects). Of these, a total of 20,426 individual object traces violate at least one part of the specified rule patterns in 4,143 apps. As an app can contain multiple errors and, by extension, multiple types of errors, the total number of apps that contain misuses is not the sum of apps that contain certain misuse types.

CogniCrypt$_{SAST}$ reports typestate errors (`ORDER` section in the rule) for 4,708 objects, and reports a total of 4,443 objects to have unsatisfied predicates (i.e., object expected a predicate from another object as per the `REQUIRES` block of a rule). The analysis also discovered 97 reachable call sites that call forbidden events. The majority of object traces that violate at least one part of a CrySL rule (54.7%) contradict a parameter constraint of a rule.

Approximately 86% of constraint violations are related to `MessageDigest`. In our manual analysis (see **RQ1**), 89 of the 110 found constraint violations originated from usages of `MD5` and `SHA-1`. We expect a similar fraction to also hold for the 11,178 constraint contradictions

reported over all 10,000 apps. Many developers still use `MD5` and `SHA-1`, although both are no longer recommended by security experts [fISB17]. CogniCrypt$_{SAST}$ identifies 1,228 (10.9%) constraint violations related to `Cipher` usages. In our manual analysis, all misuses of the `Cipher` class are due to using the insecure algorithm `DES` or the `ECB` mode of operation. This result is in line with the findings of prior studies [EBFK13, SDG+14, CNKX16].

More than 75% of the typestate errors that CogniCrypt$_{SAST}$ issues are caused by misuses of `MessageDigest`. Our manual analysis attributes this high number to incorrect usages of the method `reset()`. In addition to misusing `MessageDigest`, misuses of `Cipher` contribute 766 typestate errors. Finally, CogniCrypt$_{SAST}$ detects 157 typestate errors related to `PBEKeySpec`. The `ORDER` section of the CrySL rule for `PBEKeySpec` requires calling `clearPassword()` at the end of the lifetime of a `PBEKeySpec` object. We manually inspected 3 of the misuses and observed that the call to `clearPassword()` is missing in all of them.

Predicates are unsatisfied when CogniCrypt$_{SAST}$ expects the interaction of multiple object traces, but is not able to prove their correct interaction. With 4,443 unsatisfied predicates reported, the number may seem relatively large, yet one must keep in mind that unsatisfied predicates accumulate transitively. For example, if CogniCrypt$_{SAST}$ cannot ensure a predicate for a usage of `IVParameterSpec`, it will not generate a predicate for the key object that `KeyGenerator` generates using the `IVParameterSpec` object. Transitively, CogniCrypt$_{SAST}$ reports an unsatisfied predicate also for `Cipher` objects that rely on the generated key object.

CogniCrypt$_{SAST}$ also found 97 calls to forbidden methods. Since only two JCA classes require the definition of forbidden methods in our CrySL rule set (`PBEKeySpec` and `Cipher`), we do not find this low number surprising. A manual analysis of a handful of reports suggests that most of the reported forbidden methods originate from calling the insecure `PBEKeySpec` constructors, as we explained in Section 1.1.

From the 4,349 apps that use at least one JCA Crypto API, 2,896 apps (66.6%) contain at least one typestate error, 1,367 apps (31.4%) lack required predicates, 62 apps (1.4%) call at least one forbidden method, and 3,955 apps (90.9%) violate at least one internal constraint. Ignoring the class `MessageDigest`, and hereby excluding `MD5` and `SHA-1` constraints, 874 apps still violate at least one constraint in other classes.

RQ2: Approximately 95% of apps misuse at least one Crypto API. Violating the constraints of `MessageDigest` is the most common type of misuse.

6.3.3 Performance (RQ3)

Setup

During the analysis of our dataset, we measure the execution time that CogniCrypt$_{SAST}$ spent in each of its four main phases: It constructs (1) a *call graph* using FlowDroid [ARF+14] and then runs the actual analysis, which (2) calls the *typestate analysis* and (3) *constraint analysis* as required, attempting to (4) *resolve all declared predicates*.

In Section 6.3.2, we report that CogniCrypt$_{SAST}$ finds usages of the JCA in 4,349 of all 10,000 apps in our dataset. If we include in the reporting those usages that arise from misuses within the common libraries previously excluded (see Section 6.3.2), this number rises to 8,422. We include the analysis of the libraries in this part of the evaluation because it helps evaluate the general performance of the analysis in the worst case when whole applications are analyzed.

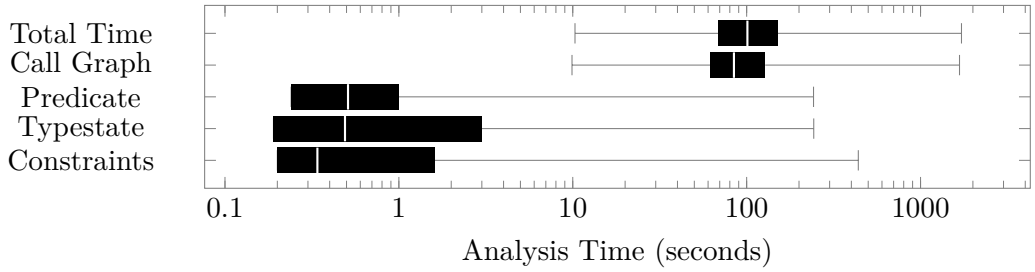

Figure 6.4: Analysis Time (in log scale) of the Individual Phases of CogniCrypt$_{SAST}$ when Running on the Apps that use the JCA.

Results

Figure 6.4 summarizes the distribution of analysis times for the four phases and the total analysis time across these 8,422 apps. For each phase, the box plot highlights the median, the 25% and 75% quartiles, and the minimal and maximal values of the distribution.

Across the apps in our dataset, there is a large variation in the reported execution time (10 seconds to 28.6 minutes). We attribute this variation to the following reasons. The analyzed apps have varying sizes—the number of reachable methods in the call graph varies between 116 and 16,219 (median: 3,125 methods). The majority of the total analysis time (83%) is spent on call-graph construction. For the remaining three phases of the analysis, the distribution is as follows. Across all apps, the resolution of all declared predicates takes approximately a median of 50 milliseconds, and the typestate-analysis phase takes a median of 500 milliseconds. The median for the constraint phase is 350 milliseconds. Therefore, the major bottleneck for the analysis is call-graph construction, a problem orthogonal to the one we address in this work. Our analysis itself is efficient and the overall analysis time is clearly dominated by the runtime of the call-graph construction.

RQ3: On average, CogniCrypt$_{SAST}$ analyzes an app in 101 seconds, with call-graph construction taking most of the time (83%).

6.3.4 Comparison to Existing Tools (RQ4)

Setup

We compare CogniCrypt$_{SAST}$ to CryptoLint [EBFK13], the most closely related tool (Section 6.6.1). Unfortunately, despite contacting the authors we were unable to obtain access to CryptoLint's implementation. We thus resort to reimplementing the original rules that are hard-coded in CryptoLint as CrySL rules. All CryptoLint rules can be modelled in CrySL. This rule set, however, still only covers a fraction of what CogniCrypt$_{SAST}$'s default CrySL rule set covers. This fact alone shows CrySL's superior expressiveness.

In this section, Ruleset$_{CL}$ denotes the set of CrySL rules we developed to model the CryptoLint rules and CogniCrypt$_{CL}$ the analysis equipped with those rules. We keep referring to CogniCrypt$_{SAST}$ as the analysis with the full rule set Ruleset$_{FULL}$.

Ruleset$_{FULL}$ consists of 23 rules, one for each class of the JCA. Ruleset$_{CL}$ comprises only six individual rules, and they only use the sections `ENSURES`, `REQUIRES` and `CONSTRAINTS`. In other words, the original hard-coded CryptoLint rules do neither comprise typestate properties nor forbidden methods. For three out of six rules, we manage to exactly capture the semantics of the hard-coded CryptoLint rule in a respective CrySL rule. The remaining three rules (3,

4, and 6 of the original CryptoLint rules) cannot be perfectly expressed as a CrySL rule, and our CrySL-based rules over-approximate them instead.

CryptoLint rule 4, for instance, requires salts in `PBEKeySpec` to be non-constant. In CrySL, such a relationship is expressed through predicates. Predicates in CrySL, however, follow a white-listing approach and therefore only model correct behaviour. Therefore, in CrySL we model the CryptoLint rule for `PBEKeySpec` in a stricter manner, requiring the salt to be not just non-constant but truly random, i.e., returned from a proper random generator. We followed a similar approach with the other two CryptoLint rules that we model in CrySL. In result, $\text{RuleSet}_{\text{CL}}$ is stricter than the original implementation of CryptoLint, producing more correct warnings. In the comparison against $\text{CogniCrypt}_{\text{SAST}}$, this setup favours CryptoLint because we assume that these additional findings to be true positives. Both rule sets are available at `https://github.com/CROSSINGTUD/Crypto-API-Rules`. Note that we have extended CrySL since the original publication of this study [KSA+18] with the built-in predicate `hardCoded` (Section 5.1.2) facilitating expressing this constraint as originally in CryptoLint. As the setup as described here favours CryptoLint, however, remodelling the corresponding CrySL rules in $\text{RuleSet}_{\text{CL}}$ such that they use the predicate would not change this study's general outcome.

Results

$\text{CogniCrypt}_{\text{CL}}$ detects usages of JCA classes in 1,866 Android apps. For these apps, $\text{CogniCrypt}_{\text{CL}}$ reports 5,507 misuses, only a third of the 20,426 misuses that $\text{CogniCrypt}_{\text{SAST}}$ identifies using $\text{RuleSet}_{\text{full}}$, our more comprehensive rule set.

Using $\text{CogniCrypt}_{\text{CL}}$, all reported warnings are related to 6 classes, compared to 23 classes that are specified in $\text{RuleSet}_{\text{full}}$. CryptoLint does not specify any typestate properties or forbidden methods. Hence, $\text{CogniCrypt}_{\text{CL}}$ does not find the 4,805 warnings that $\text{CogniCrypt}_{\text{SAST}}$ identifies in these categories using $\text{RuleSet}_{\text{full}}$. Furthermore, while $\text{CogniCrypt}_{\text{SAST}}$ reports 11,178 constraint violations with the standard rules, $\text{CogniCrypt}_{\text{CL}}$ reports only 1,177 constraint violations. Of the 11,178 constraint violations, 9,958 are due to the rule specification for the class `MessageDigest`. CryptoLint does not model this class. If we remove these violations, $\text{CogniCrypt}_{\text{SAST}}$ still reports1,609 violations, a total of 432 more than by $\text{CogniCrypt}_{\text{CL}}$.

We compare our findings to the study by Egele et al. [EBFK13] that identifies the use of `ECB` mode as a common misuse of cryptography. In that study, 7,656 apps use `ECB` (65.2% of apps that use Crypto APIs). In contrast, in our study, $\text{CogniCrypt}_{\text{CL}}$ identifies 663 uses of ECB mode in 35.5% of apps that use Crypto APIs. Although a high number of apps still exhibit this basic misuse, there is a considerable decrease (from 65.2% to 35.5%) compared to the previous study by Egele et al. [EBFK13]. We see two possible explanations that may contribute to the lower number. First, given that all apps in our study must have received an update in 2017, we believe that the decrease of misuses reflects taking software security more seriously in today's app development. Second, due to our more extensive rule set, a far greater number of apps actually counts as using cryptography, even those that do not even use `Cipher`. Hence, the ratio of crypto apps in our findings that do use `Cipher` is necessarily much smaller than for CryptoLint's original evaluation, pushing down the ratio of apps possibly containing this particular misuse.

Based on the high precision (92.6%) and recall (96.2%) values discussed in **RQ1**, we argue that $\text{CogniCrypt}_{\text{SAST}}$ provides an analysis with a much higher recall than CryptoLint. Although the larger and more comprehensive rule set, $\text{RuleSet}_{\text{full}}$, detects more complex misuses, the precise analysis keeps the false-positive rate at a low percentage.

RQ4: The more comprehensive Ruleset$_{FULL}$ detects 3× as many misuses as CryptoLint in almost 4× more JCA classes.

6.3.5 Threats to Validity

Our ruleset Ruleset$_{FULL}$ is mainly based on the documentation of the JCA [Inc17]. Although we have accumulated significant domain expertise, our CrySL-rule specifications for the JCA are only as correct as the JCA documentation. Our static-analysis toolchain depends on multiple external components and despite an extensive set of test cases, of course, we cannot fully rule out bugs in the implementation.

Java allows a developer to programmatically select a non-default cryptographic service provider. When we ran the study, CogniCrypt$_{SAST}$ did not detect such customizations but instead assumed that the default provider is used. This behaviour may lead to imprecise results because our rules forbid certain default values that are insecure for the default provider, but may be secure if a different one is chosen.

6.4 Crypto-API Misuse in Secruity-critical Android Apps

We also apply CogniCrypt$_{SAST}$ to apps from security-critical domains. By focusing the study on this type of apps, we want to validate whether the findings by previous studies [EBFK13, CNKX16, SDG+14, KSA+18] on randomly selected Android apps apply more security-critical domains, too. Specifically, the study answers the following research questions:

RQ5 To what extent does CogniCrypt$_{SAST}$ find misuses also in apps from domains with high security requirements?

RQ6 What types of misuses does CogniCrypt$_{SAST}$ find in these apps?

RQ7 How do those findings compare to those reported on randomly selected apps?

6.4.1 Setup

To answer these questions, we apply CogniCrypt$_{SAST}$, configured with Ruleset$_{FULL}$ (Section 5.3), to 250 Android apps from the domains banking, health, and password-management. On March 28th 2018, we used a Python API for the Google Play Store API[1] to compile a list of the top 50 grossing, free most-downloaded, paid most-downloaded, and trending apps from the store categories "Banking" and "Health and Fitness". Due to overlap between different top lists in each category, the merged lists amount to 162 apps. We further include into the study 88 password-manage apps. To compile popular password-management apps, we placed a search query 'password' using the same API and collected the top 88 apps on the same day as above.

We run our experiments on a 16-core virtual machine with a total of 32 GB RAM and 2 TB of disk space. For the analysis of each Android app, we grant a maximum of 8 GB of heap space. Once the call graph is constructed, verification of uses in compliance with the CrySL rule set is fast.

Many Android applications ship with similar libraries. To avoid over-reporting, we exclude findings discovered in these libraries. To this end, we restrict the reporting of errors to errors that have the same package prefix as the actual Android `apk` file.

After obtaining the results for the 250 apps, we manually analyze 25 randomly selected error reports to check for potential false positives. We collect a handful of interesting misuse examples,

[1]Python API for the Google Play Store, `https://github.com/NoMore201/googleplay-api`

Table 6.3: Types of API Misuses reported by CogniCrypt$_{SAST}$ for 250 Android apps from security-critical domains.

API Misuse Type	# Warnings	# Apps
Incorrect calling sequences	477 (35.3%)	112
Incorrect parameter values	524 (38.7%)	157
Calls to forbidden methods	17 (1.3%)	13
Insecure compositions	335 (24.8%)	105
Total	1,353	172

two of which we discuss in Section 6.4.3. We validate these findings by manually inspecting a decompiled version of the bytecode.

6.4.2 Results (RQ5 – RQ7)

In total, CogniCrypt$_{SAST}$ finds that 241 out of the 250 apps actually use the JCA. In those apps, CogniCrypt$_{SAST}$ detects 1,353 vulnerabilities, where 172 apps (71%) contain at least one misuse. Table 6.3 reports, per misuse type, the number of apps that contain at least one instance of that misuse.

RQ5: For Android apps with high security demands, CogniCrypt$_{SAST}$ finds an average of more than 5 misuses per app, with at least one misuse in 71% of all apps.

CogniCrypt$_{SAST}$ detects 477 incorrect calling sequences in 112 apps. In particular, 127 warnings are due to deviations from the specified usage protocol. The remaining 350 are caused by applications that fail to use an object to completion. For instance, a `Cipher` is set up and initialized by a call to `Cipher.getInstance()` and `Cipher.init()`, but there is no call to `Cipher.doFinal()` to complete the encryption. The most commonly misused classes are `MessageDigest` and `Cipher`, with 217 and 177 findings, respectively.

Furthermore, CogniCrypt$_{SAST}$ finds 524 parameter-constraint violations, again with the classes `MessageDigest` and `Cipher` contributing the most misuses with 341 and 106 findings, respectively. Our manual investigation of the 25 app reports revealed that most constraint errors in relation to `MessageDigest` originate from uses of `MD5` and `SHA-1`. For the class `Cipher`, most violations are related to uses of `DES` and `ECB` mode. This part of our study confirms results from our previous study in 6.3 as well as other studies [EBFK13, BDA+17, SDG+14, CNKX16].

CogniCrypt$_{SAST}$ encounters only a total of 17 calls to forbidden methods in 13 apps. This low number is not surprising though, given that across all CrySL rules, only a handful methods are declared as forbidden. Of the 17 findings, 10 were insecure `PBEKeySpec` constructors. The remaining ones relate to certain `init()` methods of the `Cipher` class. We will examine an `init()`-example more closely in a case study in Section 6.5.3.

Lastly, in 335 cases, CogniCrypt$_{SAST}$ reports insecure predicate-related misuses. This number may seem high, but, as stated before, it has to be taken with a grain of salt, because, in CogniCrypt$_{SAST}$, these insecure compositions are propagated along a chain of inter-connected uses. This propagation is reflected in the numbers for classes that are typically at the end of such chains (e.g., `Cipher`, `Mac`). Such classes generally have a higher number of reported insecure compositions than classes that are mostly used at the beginning of a use-chain. We see this as a sensible way to report these misuses because each of the uses is actually insecure, even if all

errors can be corrected by one single fix. Yet, for the same reason, the number of root causes is actually lower than the reported number suggests.

RQ6: Across all analyzed Android apps, 40% of the detected misuses are due to parameter-constraint violations, while 35% are caused by incorrect calling sequences.

Discussion Regarding **RQ7**, one can draw the following conclusions: Only 71% of apps with high security demands misuse a JCA interface, compared to 95% of randomly selected apps. Yet, those apps that contain misuses, contain more than 5 misuses on average, which makes it likely that they are highly insecure. For **RQ4**, we have already shown that previous black-listing-based approaches would have detected only about one third of those misuses. This shows that the white-listing approach is really necessary to conduct a study as comprehensive as ours.

RQ7: While fewer apps with high security demands contain crypto misuses (71% compared to 95% in a random selection), those apps contain more than 5 misuses on average, rendering them highly insecure.

6.4.3 Case Studies

The aforementioned manual analysis of 25 app reports has revealed a number of vulnerabilities that CogniCrypt$_{SAST}$ is able to find due to both the comprehensive specification approach as well as state-of-the-art analysis algorithms it employs. In the following, we give two interesting examples chosen to highlight the kinds of misuses that our study reveals on top of previous ones.

Banking App

Figure 6.5 depicts a simplified code snippet from the banking app VR-Banking by Fiducia & GAD IT AG. At the time of our study, the app had been downloaded from the Google Play Store more than 1,000,000 times. In the example, CogniCrypt$_{SAST}$ finds three vulnerabilities, all related to the definition of a `PBEKeySpec` object (Line 147). The misuses are fairly similar to our motivating example in Section 1.1. One misuse that most other checkers [EBFK13, SDG+14, BDA+17] detect as well is the low iteration count of 20. As previously explained, there is no consensus on an appropriate number, but even the lowest suggestions are four-digit numbers. The second misuse is related to the salt passed to the constructor call. The salt is defined as a constant field in the same class (Line 142), although it should be random. CogniCrypt$_{SAST}$ is capable of detecting this issue thanks to its inter-procedural, field-sensitive analysis. Lastly, CogniCrypt$_{SAST}$ issues a warning for passing `pass` to the constructor as password because it is of type `String`. We have reported the misuses in a coordinated-disclosure process and they have been fixed.

Password-management App

Figure 6.6 depicts a redacted code snippet from the password-safe app Norton Identity Safe by Symantec. According to the Google Play Store, the app had been downloaded more than 500,000 times when we conducted the study. The class in the snippet contains a method `enc()` that implements an encryption using Java's `Cipher` class. The method employs `CBC` as mode of operation (Line 162), which requires a random initialization vector (IV) to be used. In the code snippet, however, the IV is defined as a hard-coded field (Line 154). CogniCrypt$_{SAST}$ detects

```
140 public class PBEExample {
141
142     private final byte[] salt = { -4, 118, -128, -82, -3, -126, -66, -18 };
143     private SecretKey sk;
144     private SecretKeyFactory skf;
145
146     public String setupKey(String pass) {
147         PBEKeySpec pbe = new PBEKeySpec(pass.toCharArray(), this.salt, 20);
148         sk = this.skf.generateSecret(pbe);
149         ...
150     }
151 }
```

Figure 6.5: Insecure setup of password-based encryption in a major banking app with >1 million downloads.

```
152 public final class ClassIVExample {
153     private SecretKey secretKey;
154     private byte[] IV;
155
156     public ClassIVExample() {
157         this.IV = new byte[]{(byte) -57, (byte) 115, (byte) 33, (byte)
                -116, (byte) 126, (byte) -56, (byte) -18, (byte) -103, (byte)
                33, (byte) -121, (byte) 60, (byte) -56, (byte) 67, (byte) 101,
                (byte) 77, (byte) -119};
158         this.secretKey = ...;
159     }
160
161     public byte[] String enc(String plain) {
162         Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
163         cipher.init(1, this.secretKey, new IvParameterSpec(this.IV));
164         return cipher.doFinal(plain.getBytes("UTF-8"));
165     }
166 }
```

Figure 6.6: Using a constant IV, found in the password-management app Norton Identity Sage with >500,000 downloads.

this misuse through Boomerang [SDAB16a], its precise field-sensitive pointer analysis. We have reported the misuse to Symantec in a coordinated-disclosure process. Symantec have since fixed the vulnerability and documented it as CVE-2018-12240 [BKS+18].

Summary These two examples are no isolated cases. In fact, we found the two patterns in at least a dozen apps (out of only 250 apps). Many of the tools we discuss in 6.6 fail to find either one or both types of misuses. Hence, these tools would find fewer misuses than CogniCrypt$_{SAST}$.

6.5 Crypto-API Misuse in Java Software

In this section, we present a large-scale study of misuses of Crypto APIs in Java applications. With the study, we wish to answer the following research questions:

RQ8 How prevalent are misuses of Crypto APIs in Java software?

Table 6.4: Types of API Misuses Reported by CogniCrypt$_{SAST}$ for Maven Central Artefacts that use the JCA.

API Misuse Type	# Warnings	# Apps
Incorrect calling sequences	8,860 (39.1%)	2,408
Incorrect parameter values	6,827 (30.1%)	3,656
Calls to forbidden methods	203 (0.9%)	130
Insecure compositions	6,774 (29.8%)	1,737
Total	22,664	7,287

RQ9 What types of misuses are present in Java software?

RQ10 How do Java and Android software compare in terms of Crypto APIs misuses?

6.5.1 Setup

To have a representative sample set of Java applications, we collect the latest versions of all artefacts on Maven Central, the world's largest code repository for Java applications. In May 2018, the index of Maven Central lists a total of 2,768,263 JAR files. We restrict our analysis to the latest version of each individual software artefact, resulting in a dataset of 204,788 JAR files that we run CogniCrypt$_{SAST}$ on with Ruleset$_{FULL}$.

We run the study on a 32-core machine with 128 GB RAM and 2 TB of disk space. We analyze 10 artefacts at a time in parallel, and grant each analysis a maximum of 10 GB of heap space. Most of the artefacts on Maven Central are libraries, which makes it difficult to pre-compute a call graph [REH+16] for an artefact. We rely on the call graph algorithm Class Hierarchy Analysis (CHA) [DGC95] that constructs an imprecise but efficient call graph that is well suited for libraries. For the artefacts that contain uses of the JCA, it takes a median of 5.4 seconds to construct the call graph and 7.3 seconds to run CogniCrypt$_{SAST}$ per application. In total, the analysis took six days to complete for the whole dataset. To answer **RQ10**, we compare the results from our study on Maven Central to those for **RQ2**.

6.5.2 Results (RQ8 – RQ10)

Table 6.4 summarizes the results of the study. CogniCrypt$_{SAST}$ finds 7,287 Java artefacts that use the JCA. Of those, 4,929 artefacts (63.0%) produce at least one warning. In total, these artefacts contain 22,664 misuses, an average of 3.1 misuses per artefact.

RQ8: CogniCrypt$_{SAST}$ finds an average of 3.1 misuses per artefact, with at least one misuse in 63% of all artefacts, resulting in an overall lower average than in our Android study.

A more detailed analysis of the results reveals that roughly 39.1% of the misuses are parameter-constraint violations. Similar to our Android study, class `MessageDigest` is the biggest source of constraint violations (4,462 misuses). The only other class that sticks out is, again, `Cipher` with 1,262 misuses. Although we have not manually analyzed a representative number of vulnerability reports from CogniCrypt$_{SAST}$ for this dataset, given the results from our manual analyses for **RQ1**, **RQ5**, and **RQ6**, we assume most of the misuses related to these two classes come from uses of `MD5`, `SHA-1`, `DES`, and `ECB`.

CogniCrypt$_{SAST}$ further observes 8,860 incorrect calling sequences, one third stemming from incorrect calls (3,085) and two thirds from incomplete uses (5,775). Again, `MessageDigest` and `Cipher` produce most of these misuses, with 4,491 and 2,193, respectively. In all 7,287 Maven artefacts that use the JCA, CogniCrypt$_{SAST}$ has encountered 203 calls to forbidden methods. Lastly, CogniCrypt$_{SAST}$ detects 6,774 insecure compositions.

RQ9: In contrast to our evaluation of Android apps, across all studied Java artefacts on Maven Central, insecure calling sequences (39.1%) contribute the most to the detected misuses, followed by insecure parameters(30.1%).

In Section 6.3, we conclude that out of the 4,071 apps that contain uses of the JCA, 95% misuse it at least once. Our results here indicate that the rate of insecure Java applications is 63% (i.e., 32 percentage points lower). CogniCrypt$_{SAST}$ has also found a lower average of misuses per application for our sample set. For Android, CogniCrypt$_{SAST}$ found 4.8 misuses per app, while here we saw an average of 3.1 misuses per app. Therefore, in terms of overall misuse, Java applications appear to contain fewer misuses, but are still somewhat insecure overall. The distribution of misuse types overall is rather similar but exhibits one remarkable difference. That is, CogniCrypt$_{SAST}$ finds many more applications with incorrect parameters for Android apps (93.7% vs. 50.1%). For the rest, the numbers are closer to each other. There are slightly more with insecure compositions (26.5% vs. 23.8%) and incorrect calling sequences (43% vs. 33.0%), as well as slightly fewer calls to forbidden methods (1.3% vs. 1.7%).

RQ10: Comparing our answers to **RQ8** and **RQ9** with those to **RQ2**, we first observe a 34% lower rate of crypto-misusing artefacts in Maven Central than crypto-misusing Android apps in the Google Play Store. The distribution is generally rather similar, only the much lower number of apps with parameter-constraint errors is notable.

6.5.3 Case Studies

We next take a closer look at three vulnerabilities that CogniCrypt$_{SAST}$ detects. These cases CogniCrypt$_{SAST}$ is only able to uncover thanks to its white-list approach and extensive analysis. We have encountered these examples when cross-checking some of the findings.

Kerberos Application

We first discuss an example from an artefact implementing the kerberos protocol developed by a widely known vendor. The code snippet in Figure 6.7 contains two misuses. First, a `Cipher` object is instantiated for an encryption with the broken algorithm `RC4` (Line 168). Second, Line 181 in the method `calculateIntegrity()` defines a `MAC` object. This statement is followed by a call to `Mac.doFinal()`. When executed, this method will throw an `IllegalStateException` because any `MAC` object must be initialized by a call to `init()` before calling `doFinal()` on it. This misuse not only makes the code non-functional, but also insecure as a security-critical operation, namely mac-ing of data, can never be performed.

Application Server

Figure 6.8 depicts another interesting example from a popular application-server artefact. The method `getStore()` defines a `KeyStore` object and subsequently calls `load()` on it. The method

```
167 public byte[] processCipher(boolean isEncrypt, byte[] data, byte[]
        keyBytes) {
168     Cipher cipher = Cipher.getInstance("ARCFOUR");
169     SecretKey key = new SecretKeySpec(keyBytes, "ARCFOUR");
170
171     if (isEncrypt) {
172         cipher.init(Cipher.ENCRYPT_MODE, key);
173     } else {
174         cipher.init(Cipher.DECRYPT_MODE, key);
175     }
176     return cipher.doFinal(data);
177 }
178
179 public byte[] calculateIntegrity(byte[] data, byte[] key, KeyUsage usage) {
180     try {
181         Mac digester = Mac.getInstance("HmacMD5");
182         return digester.doFinal(data);
183     } catch (NoSuchAlgorithmException nsae) {
184         return null;
185     }
186 }
```

Figure 6.7: An Example Illustrating the Use of the Insecure `RC4` and Missing the Initialization of a `MAC` Object.

`KeyStore.load()` receives a password as `char[]`. This password should not be of type `String`, but in the code snippet it is. However, what is interesting about this example is what CogniCrypt$_{SAST}$ finds in addition to the wrong type for the password. The method `getStore()` is called by the method `getTrustStore()` (Line 197), which in turn retrieves the password by calling `getTrustStorePassword()` (Line 195). This method attempts to read the password from a configuration file and, if that fails, from a system property. If both attempts fail, the method calls yet another method: `getKeyStorePassword()` (Line 219). Within this method, the same config file is read twice in an attempt to retrieve the password. If both also fail, the hard-coded string `"changeit"` is returned as the password. Putting all of this together, under certain circumstances, the password used to load the keystore may not only be of type `String`, while it should not, but it may be a hard-coded string. CogniCrypt$_{SAST}$ finds this misuse, primarily because of its comprehensive CrySL rule set. On top of that, CogniCrypt$_{SAST}$ displays the password in the respective vulnerability report. This behaviour is mostly due to Boomerang [SDAB16a] that enables CogniCrypt$_{SAST}$ to retrieve the original allocation site of the password even across several methods.

Data-Visualization Application

Lastly, we discuss a misuse in the code snippet in Figure 6.9. As mentioned before, CrySL mostly follows a white-listing approach, except that it also allows for the declaration of forbidden methods. Certain `init()` methods of class `Cipher` are one instance of those forbidden methods. These `init()` methods do not allow one to pass IVs or similar extra parameters, which are, however, necessary if one wishes to use a mode of operation other than `ECB`. Since the proper generation of an IV can be tricky, the standard provider `SunJCE` can automatically prepare an IV for the developer in case of an encryption. In turn, the developer has to retrieve the IV after the encryption and supply it to the `Cipher` object responsible for the decryption by calling an appropriate `init()` method. If no IV is provided, the statement throws an `InvalidKeyException` and is, therefore, not even executed successfully. In summary, should another mode than `ECB` be

```
187 private KeyStore getStore(String type, String path, String pass) {
188     KeyStore ks = KeyStore.getInstance(type);
189     ks.load(istream, pass.toCharArray());
190     return ks;
191 }
192
193 protected KeyStore getTrustStore() {
194         [...]
195     String truststorePassword = getTruststorePassword();
196     if ((truststore != null) && (truststorePassword != null)) {
197       ts = getStore(truststoreType, truststore, truststorePassword);
198     }
199     return ts;
200 }
201
202 protected String getKeystorePassword() {
203     String keyPass = (String)attributes.get("keypass");
204     if (keyPass == null) {
205         keyPass = "changeit";
206     }
207     String keystorePass = (String)attributes.get("keystorePass");
208     if (keystorePass == null) {
209         keystorePass = keyPass;
210     }
211     return keystorePass;
212   }
213
214 protected String getTruststorePassword() {
215     String truststorePassword = (String)attributes.get("truststorePass");
216     if (truststorePassword == null) {
217       truststorePassword =
            System.getProperty("javax.net.ssl.trustStorePassword");
218         if (truststorePassword == null) {
219             truststorePassword = getKeystorePassword();
220         }
221     }
222     return truststorePassword;
223 }
```

Figure 6.8: A Hard-coded Password (`"changeit"`, Line 205) Flows to the `KeyStore.load()` Call in Line 189.

```
224 public Cipher decrypt(byte[] secure, ExternalContext ctx) {
225     SecretKey secretKey = (SecretKey) getSecret(ctx);
226     String algorithm = findAlgorithm(ctx);
227     String algorithmParams = findAlgorithmParams(ctx);
228     byte[] iv = findInitializationVector(ctx);
229
230     Cipher cipher = Cipher.getInstance(algorithm + "/" + algorithmParams);
231     if (iv != null) {
232         IvParameterSpec ivSpec = new IvParameterSpec(iv);
233         cipher.init(Cipher.DECRYPT_MODE, secretKey, ivSpec);
234     } else {
235         cipher.init(Cipher.DECRYPT_MODE, secretKey);
236     }
237
238     [...]
239     return cipher.doFinal(secure, ...);
240 }
```

Figure 6.9: An Example Illustrating an Incorrect Call to `Cipher.init()`.

used for a decryption with a symmetric block cipher, one must not call `Cipher.init()` methods that do not take an IV. However, the code snippet in Figure 6.9 does exactly that.

Lines 225–228 retrieve a secret key, an algorithm, a mode of operation, padding scheme, and an IV from an external context. CogniCrypt$_{SAST}$ fails to determine the values precisely, so it considers all possibilities. Line 230 creates a `Cipher` object configured with the algorithm and other transformation parameters. In the subsequent lines, the method checks whether the IV is `null`. If not, the correct `init()` method is called to initialize the `Cipher` object into decryption mode using the IV. However, if it is `null`, the method calls an `init()` method that does not require an IV to be passed. The way this code is set up leaves room for two insecure situations only. First, in some cases, the transformation parameters specify `ECB` as mode of operation, which is insecure. Second, `ECB` and the `else` branch may rather be thought of as a *What if* fall-back solution. Then, this call may occur for modes that do require an IV, which may lead to the statement throwing a runtime exception. In both cases, the `decrypt()` method contains insecure or non-functional code.

6.6 Related Work

We now contrast CogniCrypt$_{SAST}$ with other approaches for detecting and studies on misuses of Crypto APIs. We also relate our work to two program-repair tools targeting Crypto APIs. We conclude that existing tools for detecting and repairing misuses of Crypto APIs largely are limited mainly because they have hard-coded rule sets, and support for the most part lightweight syntactic analyses.

6.6.1 Detecting Misuses of Crypto APIs

Java and Android

More than half a dozen approaches address the detection of misuses of *Crypto* APIs, specifically. CryptoLint [EBFK13] performs a lightweight syntactic analysis to detect violations of exactly six hard-coded usage rules for the JCA in Android apps. Those six rules, while important to obey for security, resemble only a tiny fraction of the rule set we provide in this work. It is also hard to specify and validate new rules using CryptoLint, because they would have to be

hard-coded. Unlike CryptoLint, CrySL is designed to allow crypto experts to also express comprehensive and complex rules with ease. In Section 6.3, we have extensively compared our tool CogniCrypt$_{SAST}$ to CryptoLint empirically.

Another tool that finds misuses of Crypto APIs is Crypto Misuse Analyzer (CMA) [SDG+14]. Similar to CryptoLint, CMA's rules are hard-coded, and its static analysis is rather basic. Many of CMA's hard-coded rules are also contained in the CrySL rule set that we provide. Unlike CogniCrypt$_{SAST}$, CMA has been evaluated on a small dataset of only 45 apps.

Chatzikonstantinou et al. [CNKX16] manually identified misuses of Crypto APIs in 49 apps and then verified their findings using a dynamic checker. All three studies concluded that at least 88% of the studied apps misuse at least one Crypto API.

Nguyen et al. [NWA+17] present Fixdroid. The static-analysis plugin for Android Studio comes equipped with 13 rules related to security APIs. In terms of Crypto APIs, it also covers about the same rules as CryptoLint.

Rahaman et al. [RXA+19] develop the static-analysis tool CryptoGuard. Through demand-driven program slicing, a context- and flow-sensitive data-flow analysis and a number of results refinements, CryptoGuard manages to efficiently detect Crypto-API misuses. The authors have also created a benchmark to test their tool and compare it to CogniCrypt$_{SAST}$. In particular in terms of memory consumption and performance, CryptoGuard trumps CogniCrypt$_{SAST}$. However, the tool's hard-coded rule set is again rather limited with a list of fifteen misuses. While it covers four types of misuses CogniCrypt$_{SAST}$ does not, it also misses a large variety. As CogniCrypt$_{SAST}$—within their study—reports several false positives and misses actual misuses, we have investigated the benchmark and added the test cases to our test suite for CogniCrypt$_{SAST}$[2] or plan to do so for issues we deem less practically relevant.[3] While investigating the benchmark, we came to the conclusion that it is in large parts not representative of what is found in practice, based on the experience of real-word cryptographic applications the author of this thesis has accumulated in both the studies for this work (Sections 6.3 – 6.5) as well as previous work [NKMB16, NK16]. The authors report a number of false positives for CogniCrypt$_{SAST}$, but those appear to largely emerge from test cases that seem not relevant, including some that use obscure data structures such as maps to store passwords and seeds. The authors also make multiple false claims about CogniCrypt and CrySL. First, they claim CogniCrypt was mainly designed for the purpose of a user study, which is false. Their claim originates from two misunderstandings. The authors do not distinguish between CogniCrypt$_{SAST}$ and CrySL, one being the analysis, the other being a specification language. They also do not make the connection of CogniCrypt and CogniCrypt$_{SAST}$ being related although we describe their relationship in the implementation section of the original paper [KSA+18]. Second, they claim the CrySL rules prohibiting a password from having any data flows from a String object would prevent CogniCrypt$_{SAST}$ from detecting hard-coded passwords. These two rules are, however, orthogonal. Our CrySL rule set covers both types of misuses. Third, they claim that CogniCrypt$_{SAST}$ does not detect insecure uses of pseudo-random number generators, but it does. This misconception likely stems from a test case that appears to flag all uses of `java.util.Random`, instead of just those in crypto contexts.[4] Fourth, they claim CogniCrypt$_{SAST}$ reports non-crypto issues like unused variables. CogniCrypt$_{SAST}$ does not.

Muslukhov et al. [MBB18] shed light on the issue of over-reporting Crypto-API misuse in Android. To this end, they develop the tool BinSight, a static analyzer that finds the same six misuses as CryptoLint [EBFK13], however, also identifies whether a misuses occurs in one of

[2] `https://github.com/CROSSINGTUD/CryptoAnalysis/issues/134`

[3] `https://github.com/CROSSINGTUD/CryptoAnalysis/issues/165`

[4] `https://github.com/CryptoGuardOSS/cryptoapi-bench/blob/master/src/main/java/org/cryptoapi/bench/untrustedprng/UntrustedPRNGCase1.java`

the libraries an app is using or in app code itself. BinSight detects third-party code through namespace mapping. It counts any namespace not matching the app's name as third-party code. Surprisingly, package-name obfuscation only poses insurmountable problems in 2.5% of cases. The authors apply BinSight to around 132,000 apps and find that at least 70% of misuses stem from libraries misusing Crypto APIs rather than the apps themselves. In contrast to their work, in our large-scale Android analysis, we have resorted to excluding results from a list of common packages since these included the vast majority of the third-party-code findings we could identify. However, given Muslukhov et al.'s [MBB18] results, future work should investigate how many of CogniCrypt$_{SAST}$'s findings also reside in library rather than app code. Apart from that, we note that CogniCrypt$_{SAST}$, again, covers a wider variety of misuses than BinSight. Nonetheless, they were able to also observe a significant decrease of uses of ECB mode since CryptoLint's publication [EBFK13].

Braga et al. [BDA+17] present a comparative survey of free static analyzers that check for misuses of crypto APIs. The studied tools include FindSecBugs [Art18], VisualCodeGrepper [NCC18], Xanitizer [Rig], sonar-scanner [Son17], and Yasca [Sco18]. To evaluate these tools, the authors compile a benchmark of 384 test cases, 202 of which contain crypto misuses. When applying each tool to their benchmark, they find the general coverage of crypto misuses to be rather low. Xanitizer—the best among the selected—only finds 68 misuses while producing 40 false positives. The tools mostly cover simple misuses such as outdated algorithms or ECB mode, but fail on more complex cases like detecting improper IVs.

Other work has investigated other kinds of security APIs. Fahl et al. [FHM+12] analyzed 13,500 Android apps with their static checker Mallodroid. Mallodroid evaluates apps in terms of insufficient validation of TLS certificates. From their sample set, 1,074 apps do prove to fall short in that regard, leaving them vulnerable to person-in-the-middle attacks. Similarly, Georgiev et al. [GIJ+12] achieve similar results in an in-depth analysis of how a number of high-profile apps handle TLS-certificate validation.

Other Languages

Some work has also targeted Crypto APIs in languages other than Java. Li et al. [LZLG14] develop iCryptoTracer, a hybrid analysis for finding misuses of Crypto APIs in Objective-C. iCryptoTracer supports three misuses relating to symmetric encryption. They evaluation on 98 iOS apps reveals 64 to contain at least one misuse.

Feichtner [Fei19] presents the first study directly comparing cryptographic misuse on iOS and Android. To this end, they select 775 apps that are available on both platforms to investigate whether both versions contain the same or similar misuses. They also check for the six rules defined by Egele et al. [EBFK13]. For Android, they implement a custom static analyzer that uses program slicing to identify relevant lines in the code. For their iOS analysis, they employ an analyzer for finding misuses of Crypto APIs by the Institute of Applied Information Processing and Communications (IAIK) in Graz [oAIPI]. Results show that 78% of iOS and 69% of Android apps exhibit misuses.

Gu et al. [GWL+19] conduct a manual in-depth investigation on 830 misuses of Crypto APIs in Open-Source C programs. They find that almost all bugs only spread over at most ten lines. Using the results of their manual analysis as a ground truth, they also compare three static checkers. They find that all three find different misuse types, but not one finds all. The tools also generally tend to skew conservative in their analysis in favour of higher precision and to the detriment of recall.

Zhang et al. [ZCD+19] identify misuse of Crypto APIs on iot devices. Due to the diverse nature of iot devices, their analyzer CryptoREX is able to transform binary code of multiple architecture into a common IR. It then identifies relevant API calls and tracks their inputs

through the program in a backward taint analysis. In a last step, CryptoREX is checking the inputs for the same violations as Egele et al. [EBFK13]. Their evaluation on 521 state-of-the-art iot firmware shows that about 24% of firmware images misuse Crypto APIs.

Wang et al. [WLZ+17] present NativeSpeaker, a tool that checks for crypto misuses in native code. The tool can detect two kinds of crypto uses. First, it detects when native code calls the JCA (whose interfaces are implemented in plain Java). Second, it applies heuristics comprising filters on an operation's type and name to find cryptography within the native code itself. For each use found, it checks for a number of misuse types related to symmetric encryption only. In this context, NativeSpeaker finds uses of outdated crypto algorithms, uses of ECB mode, and improper key material.

Mitchell and Kinder [MK19]'s approach to detect misuses of Crypto APIs allows for annotating JavaScript Crypto libraries. These annotations lead the library to report to its user when it is being misused. They provide a proof-of-concept implementation of a Crypto library in JavaScript.

Leuer [Leu19] proposes Sharper Crypto-API Analysis, an analyzer reports misuses of C# Crypto APIs. To this end, the author first analyzes extensively both the requirements for such a tool as well as the environment that C# and, by proxy, .Net offer in terms of cryptography. Sharper Crypto API Analysis covers a list of twelve hard-coded misuses, ranging from low iteration count for PBE, insufficient salt, and the use of ECB mode or insecure cryptographic algorithms. Leuer has integrated Sharper Crypto-API Analysis into Visual Studio.

Summary

None of the previous approaches facilitate rule creation by means of a higher-level specification language. Instead, the rules are hard-coded into each tool's code, making it hard for non-experts in static analysis to extend or alter the rule set. Consequently, the tools are not completely incapable of supporting CogniCrypt$_{SAST}$'s broad range of misuses, but extending one to do so requires a) intricate knowledge of the respective tool and its code and b) a lot of repetitive manual implementation work. This limitation also makes it impossible to share rules among tools. Moreover, such hard-coded rules are quite restricted, causing the tools to have a very low recall (i.e., missing many actual API misuses). Thanks to CrySL, on the other hand, CogniCrypt$_{SAST}$ can cover a much wider range of violations. By defining crypto-usage rules in CrySL instead of hard-coding them, non-experts can easily configure CogniCrypt$_{SAST}$. The rules aso become reusable in other contexts. This flexibility further becomes important when confronted with different APIs, multiple versions of an API, or standards one wants to check a program for compliance with. NIST [GGF17], the BSI [fISB17], FIPS [Lab19], and other cryptographic standards may overlap in certain requirements, but they also differ in others. Such variability can hardly be supported by analysis tools with hard-coded rules.

6.6.2 Repairing Misuses of Crypto APIs

Two tools go beyond detecting misuses and attempt to fix them as well. CDRep [MLLD16] applies a by-one-rule-extended version of CryptoLint to a target program. For each of the seven kinds of misuses CryptoLint finds, the authors have devised a fix template. In a second phase, CDRep applies this fix by instrumenting the program's bytecode. In an evaluation on 8,592 Android apps, the tool manages to repair around 95% of the misuses it has detected. FireBugs [SZSS19] pursues a similar goal. The tool's authors have defined code patterns that contain API misuses. Bootstrapped with these patterns, FireBugs analyzes a target program through program slicing, and repairs it using a series of AST operations. To finally apply a patch, FireBugs employs aspect-oriented programming to weave it into the target program.

In contrast, CogniCrypt$_{SAST}$ does not repair the target program, but only analyzes it. There is however ongoing work on such a program-repair tool on top of CrySL and CogniCrypt$_{SAST}$ that combines CrySL's large coverage, CogniCrypt$_{SAST}$'s precise analysis, and a means to repair vulnerable software automatically. We discuss further the ideas behind this tool in Section 9.2, the approach itself is, however, beyond the scope of this thesis.

6.7 Conclusion

In this chapter, we have presented CogniCrypt$_{SAST}$, a static analysis checking for compliance of a target program with a set of CrySL rules. Bootstrapped with the Ruleset$_{FULL}$ CrySL rule set, this approach is capable of detecting misuses of Crypto APIs in Java programs. To this end, CogniCrypt$_{SAST}$ employs efficient and precise state-of-the-art data-flow analysis. Applying CogniCrypt$_{SAST}$, the analysis for our extensive ruleset Ruleset$_{FULL}$, to 10,000 Android apps, we found 20,426 misuses spread over 95% of the 4,349 apps using the JCA. Similarly, we applied CogniCrypt$_{SAST}$ to 2,700,000 artefacts on Maven and it detected misuses in 63% of the artefacts that use cryptography. CogniCrypt$_{SAST}$ is also highly efficient: it analyzed all of Maven Central in under a week and for more than 75% of the apps the analysis finishes in under 3 minutes, where most of the time is spent call graph construction. When analyzing more security-critical apps, we unfortunately found there to still be many misuses (1,353 misuses in 172 apps) and many apps with misuses (71%).

There are several ways forward for CogniCrypt$_{SAST}$. The issue of over-reporting raised by Muslukhov et al. [MBB18] should be investigated and addressed. The valid imprecision issues raised by Rahaman et al. [RXA+19] deserve addressing as well. Lastly, future work should also explore evaluating CogniCrypt$_{SAST}$'s reports. If users fail to understand the warnings CogniCrypt$_{SAST}$ reports to them, the tool serves little purpose.

7

CogniCrypt$_{\text{gen}}$

In this chapter, we present CogniCrypt$_{\text{GEN}}$. Previous attempts at solving cryptographic misuse have largely focused on misuse detection (Section 6.6) or less direct means than tool support (Chapter 4). Java developers do, however, also request more direct support [NKMB16]. By presenting CogniCrypt$_{\text{GEN}}$, our second tool support built on top of CrySL, we fill this gap. CogniCrypt$_{\text{GEN}}$ is a code generator for secure integrations of Crypto APIs. The tool operates on a Java project into which it generates code and accepts as input a template with interface and glue code, as well as usage rules in CrySL. Our design of CogniCrypt$_{\text{GEN}}$ simplifies the used code templates, causing the vast majority of the code to be generated fully automatically from CrySL definitions. This code is provably correct by construction and secure (assuming a correct and secure specification of CrySL rules by domain experts).

We evaluate CogniCrypt$_{\text{GEN}}$ along four different dimensions. First, to determine its expressiveness, we implement templates for eleven common cryptographic use cases in CogniCrypt$_{\text{GEN}}$. Second, we measure runtime performance and memory consumption of CogniCrypt$_{\text{GEN}}$ on all those use cases. We then compare CogniCrypt$_{\text{GEN}}$ to CogniCrypt$_{\text{OLD-GEN}}$, CogniCrypt$_{\text{GEN}}$'s predecessor within the context of CogniCrypt's prototypical implementation as an Eclipse plugin. We describe CogniCrypt$_{\text{OLD-GEN}}$ in much more detail in Sections 7.3 and 7.4. In this context, we compare the effort to create and maintain the artefacts for the eight use cases that CogniCrypt$_{\text{OLD-GEN}}$ and CogniCrypt$_{\text{GEN}}$ have in common in terms of the artefacts that are required to implement them in each of the two code generators. Lastly, we investigate the usability of CogniCrypt$_{\text{GEN}}$ for a Crypto-API developer, i.e., a domain expert who may integrate new use cases or modify existing ones. To this end, we conduct a user study with 16 participants, for which we ask them to perform a series of modifications to artefacts of existing use-case implementations. In conclusion, we manage to implement all common use cases we find in CogniCrypt$_{\text{GEN}}$. CogniCrypt$_{\text{GEN}}$ further outperforms CogniCrypt$_{\text{OLD-GEN}}$ in terms of usability and maintainability, while showing negligible memory overhead and fast performance results of below ten seconds.

7.1 Generating Secure Code From CrySL

In this section, we present CogniCrypt$_{\text{GEN}}$. First, we discuss the goals and considerations of our design. We then go further into its API, the code templates that interface CogniCrypt$_{\text{GEN}}$, and the underlying code-generation algorithm.

```
241 public SecretKey generateKey(char[] pwd) {
242   byte[] salt = new byte[32];
243   javax.crypto.SecretKey encryptionKey = null;
244
245   CrySLCodeGenerator.getInstance().
246     includeClass("java.security.SecureRandom").addParameter(salt,"salt").
247     includeClass("java.security.PBEKeySpec").addParameter(pwd,"password").
248     includeClass("javax.crypto.SecretKeyFactory").
249     includeClass("java.security.SecretKey").
250     includeClass("javax.crypto.SecretKeySpec").addReturnObject(encryptionKey).
251     generate();
252   return encryptionKey;
253 }
```

Figure 7.1: CogniCrypt$_{GEN}$ Template that Generates a Correct Java Implementation for PBE from Figure 1.1.

7.1.1 Design Considerations

CogniCrypt$_{GEN}$ targets a very practical problem: solving the widespread misuse of Crypto APIs through code generation. To this end, we put usability at the center of the design effort. Usability first and foremost refers to usability for regular Java developers. To truly help those developers, CogniCrypt$_{GEN}$ must support the most common cryptographic use cases. It must also integrate into the development workflow of a regular developer, which puts limitations on the running time of CogniCrypt$_{GEN}$. On top of these aspects, prioritizing usability, for us, also means to have a usable development and maintenance process of the artefacts in the backend of CogniCrypt$_{GEN}$ (i.e., CrySL rules and code templates). The target audience of CogniCrypt$_{GEN}$'s backend are crypto experts who have either developed a library that they want to integrate into CogniCrypt$_{GEN}$ or have some experience with one. To keep things simple, we opt for Java code templates as opposed to other template languages that Java crypto experts are not likely familiar with. Since CrySL also has a Java-like syntax, developers are not required to familiarize themselves with the syntax of other languages. While CrySL is an additional artefact that may not be needed by other template engines in particular, or code-generation approaches in general, developing rules in CrySL comes with the additional advantage of gaining other tool support from the CogniCrypt ecosystem. The result of our design is a code generator combining code templates with CrySL rules.

7.1.2 Configuring Solutions with Java Code Templates

CogniCrypt$_{GEN}$'s code templates are regular Java classes. They allow crypto experts to (1) include use-case-specific boilerplate code, (2) specify CrySL rules that make up the given use case, and (3) pass objects from the boilerplate code to CogniCrypt$_{GEN}$. Figure 7.1 shows the code template for implementing the PBE example from Figure 1.1 *correctly and securely*. Figure 7.2 shows how the generated code uses this template. Line 242 in the template defines a `salt`. This explicit definition is necessary because the involved APIs require a byte array, but do not create one themselves. The line after, Line 243, defines a cryptographic key called `encryptionKey`, which the generated code uses to store the generated key (Line 268 in Figure 7.2).

Starting at Line 245, the template calls CrySL rules and instantiates their parameters using a *fluent API* [Fow05]. The call to `getInstance()` instantiates the code generator. Line 246 includes the class `java.security.SecureRandom` into the code generation through a call to `includeClass()`. In the next line, the call to `addParameter()` associates the byte array `salt` in the template with the variable `salt` in the CrySL rule for `SecureRandom`. Lines 246–247

```
255 public class TemplateClass {
256   public SecretKey generateKey(char[] pwd) {
257     byte[] salt = new byte[32];
258     SecretKey encryptionKey = null;
259
260     SecureRandom secureRandom = SecureRandom.getInstance("SHA1PRNG");
261     secureRandom.nextBytes(salt);
262     PBEKeySpec pBEKeySpec = new PBEKeySpec(pwd, salt, 27799, 128);
263
264     SecretKeyFactory secretKeyFactory = SecretKeyFactory.getInstance
265           ("PBEWithHmacSHA512AndAES_128");
266     SecretKey secretKey = secretKeyFactory.generateSecret(pBEKeySpec);
267     byte[] keyMaterial = secretKey.getEncoded();
268     encryptionKey = new SecretKeySpec(keyMaterial, "AES");
269
270     pBEKeySpec.clearPassword();
271     return encryptionKey;
272   }
273 }
274
275 public class OutputClass {
276   public void templateUsage(char[] pwd) {
277     TemplateClass tc = new TemplateClass();
278     SecretKey key = tc.generateKey(pwd);
279   }
280 }
```

Figure 7.2: The Code that CogniCrypt$_{GEN}$ Generates Using the Template from Figure 7.1.

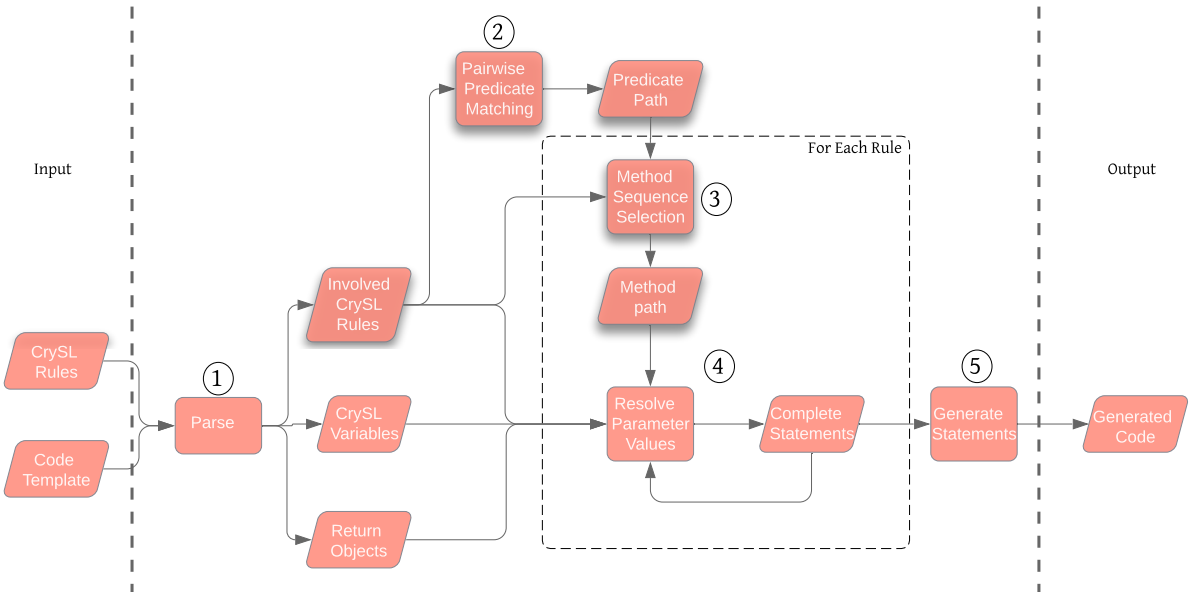

Figure 7.3: The General Workflow of CogniCrypt$_{GEN}$.

properly randomize `salt` before using it during key generation. Lines 247–250 then create a `java.security.PBEKeySpec` key and transform it into a `javax.crypto.SecretKeySpec` key. Finally, the call to `addReturnObject()` assigns `encryptionKey` the role of a return object. During the generation, the return value of the constructor of `javax.crypto.SecretKeySpec` is stored in `encryptionKey`. CogniCrypt$_{GEN}$ selects the constructor because it is the last method of that class that needs to be called according to the CrySL rule. The password-based key is thus assigned to `encryptionKey`. Line 252 returns the key as part of the boilerplate code.

7.1.3 Generating Secure Code from Templates

Figure 7.3 shows the workflow of CogniCrypt$_{GEN}$. In the following, we describe how it works and will refer to the individual steps using their corresponding numbers in the figure. For each call to the code-generator API in a template, CogniCrypt$_{GEN}$ first collects all rules and their parameters from an API call chain ①. The chain in Line 245 in method `generateKey()` requires rules for `java.security.SecureRandom`, `java.security.PBEKeySpec`, `javax.crypto.SecretKeyFactory`, and `javax.crypto.SecretKeySpec`. In addition, the CrySL rules for `java.security.SecureRandom` and `java.code.PBEKeySpec` get attached the objects `salt` and `pwd` to their in-rule variables `salt` and `password`, respectively. Lastly, the rule for `SecretKeySpec` yields the object `encryptionKey` as a return object.

CogniCrypt$_{GEN}$ then iterates through the rules to assemble a list of predicates that link rules to one another ②. These links form a path that CogniCrypt$_{GEN}$ uses to select appropriate method sequences for a given class ③. If two classes are connected through a predicate, CogniCrypt$_{GEN}$ may, for the class that should *ensure* the predicate, only select method sequences that eventually grant this predicate. Similarly, for the class that *requires* the predicate, CogniCrypt$_{GEN}$ picks method sequences that make use of the predicate. For the PBE example in Figure 7.1, `PBEKeySpec` can generate the predicate `speccedKey` on itself. `SecretKeyFactory`, in turn, requires this predicate on the key specification object that it uses to generate a key. Hence, CogniCrypt$_{GEN}$ connects both rules using the predicate `speccedKey` on the `PBEKeySpec` object. If CogniCrypt$_{GEN}$ were unable to establish a path between `PBEKeySpec` and `SecretKeyFactory`, it would not have taken the former into account when generating code for the latter.

Next, CogniCrypt$_{GEN}$ iterates through all rules again to assemble the code, which includes both generating method calls ③ for all involved classes and finding appropriate values for their parameters ④. For each CrySL rule, CogniCrypt$_{GEN}$ first compiles a list of correct paths of method calls according to the specified usage pattern in the `ORDER` section ③. To this end, CogniCrypt$_{GEN}$ translates a rule's pattern into a finite state machine. The tool then classifies any path of method calls that leads to an acceptable state in the state machine as correct. When assembling such paths, CogniCrypt$_{GEN}$ has to deal with methods that, according to the state machine, may be called multiple times. CogniCrypt$_{GEN}$ translates such methods into two different paths: one where the method is not called and one where it is. CogniCrypt$_{GEN}$ does not currently support repeated calls. However, in our experiments with the JCA, this lack of support has not proven to be a problem. In scenarios where more than one correct path is found, CogniCrypt$_{GEN}$ applies a set of filters to reduce the number of sequences. Paths that do not include objects required by the code template through calls to `addParameter()` cannot implement the use case and are, therefore, eliminated. For `PBEKeySpec`, its CrySL rule in Figure 5.2 prescribes one specific constructor (`create`) and the method `clearPassword()` to be called in that order. Therefore, for this class, CogniCrypt$_{GEN}$ finds only this one possible path to an accepting state. Similarly, CogniCrypt$_{GEN}$ discards paths that may lead to predicates different than the ones associated with it. In the case of `PBEKeySpec`, the only possible path grants the correct predicate `speccedKey`. Consequently, CogniCrypt$_{GEN}$ does not remove this

path from the list of possible paths to take. Instead, CogniCrypt$_{GEN}$ generates the call to the `PBEKeySpec` constructor in Line 262 of Figure 7.2.

The second call, the call to `clearPassword()`, CogniCrypt$_{GEN}$ appends to the block of API statements in Line 270 before the return statement because the method invalidates the `speccedKey` predicate of the object. Instead of generating calls to such invalidating methods directly, CogniCrypt$_{GEN}$ collects them and generates them at the end of the method.

For each method call on the remaining paths, CogniCrypt$_{GEN}$ applies several heuristics to resolve possible parameter values ④. First, it attempts to match parameters required by the template through calls to `addParameter()` to a given parameter in a method call. Two objects match when the CrySL variable mapped to an object in the `addParameter()` call in the template is the one used in the method call. At Line 262 of Figure 7.2, CogniCrypt$_{GEN}$ matches `password`, the first parameter in the constructor `create` of `PBEKeySpec`, to the argument `pwd` of the method `generateKey()`. It does so because the call to `addParameter()` maps `pwd` to the CrySL variable `password` in `PBEKeySpec` (Line 247). In case of no match, CogniCrypt$_{GEN}$ further attempts to match a parameter to objects in the generated code that have received a matching predicate. For the PBE example, CogniCrypt$_{GEN}$ generates code for `SecureRandom` that ensures the predicate `randomized` for `salt`. CogniCrypt$_{GEN}$ then matches the second parameter in the call to the `PBEKeySpec` constructor to this `salt` object because it has the same type and requires the same predicate. CogniCrypt$_{GEN}$ conducts one further step to resolve remaining method parameters that may not be matched to any existing objects from the code template or the generated code. For those, it queries constraints from the respective CrySL rule and fetches secure values from the first appropriate constraint that it finds. For value constraints of the form `var in {Literal1, ..., LiteralN}`, it selects the first option `Literal1`. This type of constraint usually comes into play for algorithms and key sizes (e.g., in `KeyGenerator` or `Cipher` [KSA+19a]). From a security perspective, since all values in a CrySL rule ought to be correct, it does not matter which value CogniCrypt$_{GEN}$ chooses. Another type of constraints becomes relevant for the third parameter of the `PBEKeySpec` constructor. According to the rule in Figure 5.2, the iteration count must be $\geq$ 10,000. CogniCrypt$_{GEN}$ generates the closest value that satisfies this constraint, which is 10,000 (Line 262 in Figure 7.2).

For cases in which this attempt to resolve the parameter fails as well, we decided to prioritize compilability of the generated source code over completeness. That is, CogniCrypt$_{GEN}$ adds the unresolvable parameter to the wrapper method that the call belongs to. During the development of our code templates, this feature has proven useful for debugging. We have first specified which CrySL rules should be included in the generation and after running CogniCrypt$_{GEN}$, the generated code showed which parameters needed additional specification. We believe this feature will also help crypto experts in similar situations. For the final code template, however, this step is meant as a fallback solution because it changes the wrapper method as defined in the code template and de-facto complicates the use of the method. In practice, CogniCrypt$_{GEN}$ does not have to take this final step for any of the use cases we have implemented.

If, at the end of this process, CogniCrypt$_{GEN}$ needs to choose between multiple method paths with fully-resolvable parameters, it selects the shortest one. That is, CogniCrypt$_{GEN}$ opts for the method path with the fewest method calls as well as the smallest number of parameters. When all calls are selected and all parameter objects have been assigned values, CogniCrypt$_{GEN}$ generates the produced code into the target program ⑤. This process is then repeated for all calls to CogniCrypt$_{GEN}$'s fluent API within a given template. Once CogniCrypt$_{GEN}$ has processed a template, it also generates a method that showcases the usage of the generated code. To this end, it creates a new class and method, in which it first instantiates an object of the template class. For our running example, Line 276 in Figure 7.2 marks the first statement of the corresponding method. In that method, CogniCrypt$_{GEN}$ iterates through all methods

of the template class that contains calls to the fluent API and generates calls for them. For `generateKey()` in the template, the generated call is at Line 278. Other methods are assumed to be internal helper methods. CogniCrypt$_{GEN}$ further attempts to match parameters by checking if a given parameter's type matches that of any return values of previous calls. To ensure compilability, CogniCrypt$_{GEN}$ pushes up parameters where no matching is possible, e.g., `pwd` of `generateKey()`, to become parameters of method `templateUsage()`. For `pwd`, this refactoring produces indeed the correct result, as the password should be an input rather than a hard-coded value. In conclusion, we view this method as useful for developers, because they do not need to engage with the generated code, but only with this summary method. We drew inspiration for this feature from CogniCrypt's previous code-generator CogniCrypt$_{OLD\text{-}GEN}$, in which such a method is hard-coded into the tool's templates.

7.2 Implementation Details

We developed CogniCrypt$_{GEN}$ on top of the existing infrastructure for CogniCrypt. We had to first modify CrySL with respect to the initial design. For encryption, the JCA offers one API for both asymmetric and symmetric encryption: `Cipher`. Until our work on CogniCrypt$_{GEN}$, the rule for `Cipher` had included one long list of secure algorithms indiscriminately of whether they are performing the former or the latter. For the purpose of CogniCrypt$_{SAST}$'s program analysis, such a distinction does not need to be made. However, implementing use cases involving hybrid encryption (Section 7.3) requires differentiating between them. For that purpose, we introduced a new built-in predicate `instanceof(cryslVariable, javaType)`. By means of this predicate, the CrySL rule for `Cipher` now only allows symmetric-encryption algorithms when a key used for encryption is of type `SecretKey` or subtypes (i.e., `instanceof(key, java.security.SecretKey`). Asymmetric encryption algorithms may only be used when the key is either a private or a public key, indicating that the `Cipher` object implements an asymmetric encryption. After finishing our work on CogniCrypt$_{GEN}$, we have extended CogniCrypt$_{SAST}$ to support this predicate as well[1] because it may improve its recall as the tool no longer considers certain incorrect combinations of symmetric key and asymmetric cipher (or vice versa) secure.

We further needed to modify the existing JCA rule set in the following ways. For some rules (e.g., `Signature` and `KeyGenerator`), we changed the position of arguments in the constraints `var in {Literal1, ..., LiteralN}` to better reflect the preferences in algorithm selection that CogniCrypt$_{GEN}$ should follow [GGF17, fISB17, Nat18]. We have also added a new parameter to some predicates (e.g., `Signature`), where the first parameter was not the return value of a cryptographic operation (e.g., the boolean return value of `Signature.verify()`). In all respective cases, CogniCrypt$_{GEN}$ requires the return value to store it in the correct variable as assigned in the template through a call to `addReturnObject()`.

To parse rules, we also use the CrySL parser. To parse templates, traverse rules, and modify code, we implemented our own custom solution. This solution builds on top of the Eclipse Java Development Toolkit (JDT). To parse the templates and apply changes to them in the target project, we have followed a visitor pattern using the JDT's abstract-syntax-tree (AST) APIs.

Prior to our work on CogniCrypt$_{GEN}$, in the initial phase of CogniCrypt, we had implemented CogniCrypt$_{OLD\text{-}GEN}$ as a code generator for CogniCrypt [KNR+17]. CogniCrypt$_{OLD\text{-}GEN}$ uses XSL templates to define use-case specific code and points of variability. An algorithm model in the variability-modelling language Clafer [JSM+19] supplies correct values (i.e., algorithms) based on user input [NK16]. We further compare CogniCrypt$_{GEN}$ to CogniCrypt$_{OLD\text{-}GEN}$ in Sections 7.3 and 7.4. After finishing the development of CogniCrypt$_{GEN}$, we have replaced CogniCrypt$_{OLD\text{-}GEN}$ with CogniCrypt$_{GEN}$ which implements the eight JCA use cases that

[1] Extension in commit 0971fa8 in the CogniCrypt$_{SAST}$ repository on Github.

CogniCrypt supports: password-based encryption for the data types (1) file, (2) string, and (3) byte array, hybrid encryption for the data types (4) file, (5) string, and (6) byte array, (7) digital signing, and (8) secure password storage.

7.3 Evaluation

To evaluate CogniCrypt$_{GEN}$, we aim to answer the following research questions:

RQ11 Can CogniCrypt$_{GEN}$ implement common cryptographic use cases?

RQ12 Does CogniCrypt$_{GEN}$ produce code quickly enough to be used in everyday software development?

RQ13 What is the memory consumption of CogniCrypt$_{GEN}$?

RQ14 What is the effort to create and maintain the artefacts for CogniCrypt$_{GEN}$ to implement common cryptographic use cases?

RQ15 Do contributors to CogniCrypt$_{GEN}$ perceive a usability gain compared to a state-of-the-art solution using XSL?

With the first three research questions, we aim to determine whether CogniCrypt$_{GEN}$ may actually support developers. If CogniCrypt$_{GEN}$ is incapable of implementing the most common cryptographic use cases (**RQ11**), it cannot meaningfully reduce cryptographic misuse. Similarly, if its memory consumption and runtime exceed the average capabilities of workstations (**RQ12** and **RQ13**), application developers will not use it. **RQ14** and **RQ15** then focus on crypto developers who wish to implement use cases for their own APIs. If CogniCrypt$_{GEN}$ requires too much effort (**RQ14**) or developers do not find it intuitive to use (**RQ15**), it is unlikely that a crypto developer will integrate their APIs with CogniCrypt$_{GEN}$, even if they get access to other tool support in CogniCrypt.

7.3.1 Implementation of common use cases (RQ11)

Setup

To answer **RQ11**, we have first gathered common cryptographic use cases from multiple sources. We then attempted to implement them with CogniCrypt$_{GEN}$. To check the validity of the generated code with respect to compilability and security, we have further run the Java compiler and CogniCrypt$_{SAST}$ on them.

Results

We collected eleven use cases from three different sources. Table 7.1 shows all use cases that CogniCrypt$_{GEN}$ supports as well as their respective sources. We first identified eight of eleven cryptographic use cases that CogniCrypt$_{OLD\text{-}GEN}$ [KNR+17] supports and that use the JCA. We discard the three remaining ones. No sufficiently advanced CrySL rules exist for these use cases because they are not based on the JCA. In their study, Nadi et al. [NKMB16] compiled a list of common usage scenarios by (1) analyzing the implemented use cases of 100 randomly selected GitHub projects that implement Java cryptography and (2) asking participants for cryptographic programming tasks that they commonly have to implement. We have also collected the responses often found in projects and popular with participants. Lastly, Mindermann and Wagner [MW18] have collected an online repository that aims at providing secure implementations for common cryptographic use cases. We have included this repository's use cases into our list as well.

Table 7.1: Common Cryptographic Use Cases

#	Use Case	Source	Runtime in CC_{gen} (s)	Memory Consumption in CC_{gen} (MB)
1	PBE on Files	[KNR+17]	7.0	14.1
2	PBE on Strings	[KNR+17], [MW18]	6.7	13.5
3	PBE on Byte-Arrays	[KNR+17]	7.1	66.6
4	Symmetric-Key Encryption	[MW18], [NKMB16]	6.8	6.0
5	Hybrid File Encryption	[KNR+17]	6.7	2.5
6	Hybrid String Encryption	[KNR+17]	6.6	4.2
7	Hybrid Byte-Array Encryption	[KNR+17]	6.9	56.7
8	Asymmetric String Encryption	[MW18]	6.8	34.1
9	Secure User-Password Storage	[KNR+17], [MW18]	8.1	22.7
10	Digital Signing of Strings	[KNR+17], [MW18], [NKMB16]	7.5	7.1
11	Hashing of Strings	[MW18]	6.7	14.2

We have successfully implemented all eleven use cases. The implementations of use cases 1–3 are virtually the same in CogniCrypt$_{GEN}$, because they involve the same classes, which leads to having the exact same calls to CogniCrypt$_{GEN}$'s fluent API . Only the wrapper code around the fluent-API calls changes depending on the data type (i.e., File, String, or Byte Array) that is encrypted. The same is true for use cases 5–7, which all deal with hybrid encryption but on different data types. None of the generated code snippets cause compiler errors or true misuses identified by CogniCrypt$_{SAST}$. They do cause CogniCrypt$_{SAST}$ to report a handful of false positives that are hard to avoid but can be circumvented in the context of CogniCrypt.[2]

7.3.2 Performance (RQ12 and RQ13)

Setup

To answer **RQ12**, we measure the average running time for each use case in Table 7.1. We ran each use case ten times, collect the measurements using `java.lang.System.currentTimeMillis()`, and computed the average of the measurements.

To answer **RQ13**, we ran CogniCrypt$_{GEN}$ for each task again. We capture the memory consumption of the Eclipse process through the system memory monitor both before and during CogniCrypt$_{GEN}$'s run. We then subtract the before-value from the highest value during the run. Please note, in pre-experiments, we ran several use cases multiple times, but found the fluctuation in memory usage as negligible (within half a megabyte). We hence decided to measure the memory consumption from only a single run.

We ran the experiments on a Windows 10 machine with four Intel Core i7-5600U CPUs running at 2.6GHz and 16 GB of RAM. We executed all runs on an Eclipse 2019-06 for RCP and RAP developers using Java 1.8.0_161 and gave the JVM 8 GB of RAM.

Results

We list the results for **RQ12** and **RQ13** in the last two columns of Table 7.1. CogniCrypt$_{GEN}$ takes between 6.6 and 8.1 seconds. Therefore, all runs are well below ten seconds, making CogniCrypt$_{GEN}$ easily integrable into a developer's programming workflow.

[2] `https://github.com/CROSSINGTUD/CryptoAnalysis/issues/80`

Table 7.2: Comparing the Required Lines of Code (LOC) to Implement the Use Cases of CogniCrypt$_{OLD\text{-}GEN}$ in Both CogniCrypt$_{OLD\text{-}GEN}$ and CogniCrypt$_{GEN}$.

	LOC in CogniCrypt$_{old\text{-}gen}$		**LOC in CogniCrypt$_{gen}$**
#	**XSL**	**Clafer**	**Java**
1	140	117	57
2	138	117	57
3	111	117	51
5	158	90	74
6	156	90	74
7	129	90	68
9	139	67	55
10	115	43	40

In terms of memory consumption, CogniCrypt$_{GEN}$ consumes between 2.5 and 66.6 MB on top of the regular Eclipse process. During our experiments, the latter oscillated between 900 MB and 1.2 GB of RAM. We conclude that CogniCrypt$_{GEN}$'s memory overhead is negligible.

7.3.3 Effort of Artefact Creation and Maintenance (RQ14)

Setup

To approximate the effort of rule creation and maintenance, we compare the artefacts needed to implement the eight cryptographic use cases in CogniCrypt$_{OLD\text{-}GEN}$ to their implementations in CogniCrypt$_{GEN}$. In particular, we compare the total number of lines of code a crypto expert would have to write as well as the language skills required by a developer to implement a use case using both code generators.

We have only investigated artefacts that are specific to the respective code generator. That is, for CogniCrypt$_{OLD\text{-}GEN}$, we looked at Clafer model and XSL code templates. For CogniCrypt$_{GEN}$, on the other hand, we only investigated the code templates, not the involved CrySL rules, because they are not CogniCrypt$_{GEN}$ specific, but are instead developed to receive general support for an API by CogniCrypt.

Results

Table 7.2 shows the sizes of the different artefacts for the eight use cases that CogniCrypt$_{OLD\text{-}GEN}$ supports. Overall, a developer needs to write at least 115 lines of XSL code and 43 lines of Clafer model to support any of those use cases in CogniCrypt$_{OLD\text{-}GEN}$, with the same use case only requiring 40 lines in Java for CogniCrypt$_{GEN}$. On average, each use case implements 136 lines of code in XSL and 91 lines in Clafer. In contrast, a developer needs to write an average of only 60 lines of Java code in CogniCrypt$_{GEN}$ to implement those use cases. CogniCrypt$_{GEN}$ thus has two advantages. First, artefact maintainers need to only keep track of around 25% of the lines of code. Second, crypto experts who have implemented a library in Java do not need to learn extra languages (i.e., Clafer and XSL) to implement their use cases in CogniCrypt$_{GEN}$. Instead, they may define their security code entirely in Java, a language they need to be familiar with to implement their cryptographic library in the same language. When defining code templates in an IDE such as Eclipse, the crypto experts receive the more advanced development support for Java (e.g., type checking and auto-compiling) compared to what editors or IDEs

provide for XSL or Clafer. Those advantages carry over to scenarios where experts may use CrySL in domains other than cryptography.

7.3.4 Usability (RQ15)

Setup

To answer **RQ15**, we conducted a small-scale user study with 16 participants. We recruited participants among graduate students at our local university. Each participant is given two programming tasks we describe below, both of which we based on common cryptographic use cases in Table 7.1, one with CogniCrypt$_{\text{GEN}}$ and one with CogniCrypt$_{\text{OLD-GEN}}$.

We choose to compare against CogniCrypt$_{\text{OLD-GEN}}$ for two reasons. First, it is the tool whose goal and purpose is closer to CogniCrypt$_{\text{GEN}}$'s than any other. Second, XSL, as an approach to template-based code generation, provides the ideal setting to compare against. Templates are defined in an extra language with extra features, but still providing a way to write Java code directly. Domain experts might find the additional layer of abstraction this extra language provides useful because it produces a clear cut between code template and generated code. We, however, assume this to not be the case, at least for cryptographers because, from our prior experience working with cryptographers, we can report that they often do not know any template languages. If participants of our study who by and large also lack experience with template languages nonetheless preferred CogniCrypt$_{\text{OLD-GEN}}$ despite the extra language, we would expect domain experts to also favour the old code generator rather than the new one.

Consequently, we designed the tasks such that they rely heavily on modifying code templates, instead of Clafer models and CrySL rules. Task 1, based on use case ten in Table 7.1, asks participants to (1) change a solution that hashes strings to one that hashes files and (2) fix the name of the chosen algorithm that the code generator produces. Task 2, based on use case four in Table 7.1, asks participants to (1) add proper randomization of an initialization vector for symmetric encryption and (2) prohibit the code generator from using an outdated algorithm. To avoid learning and other carry-over effects, we follow a latin-square approach [Gao05] when randomly assigning tasks and code generators to participants. We give participants 30 minutes to complete each task. Before participants start solving the tasks, the respective instructor gives a 25-minute introduction to both code-generation tools performing the same two modifications on use case eleven in Table 7.1. After participants have completed their work on the tasks, we ask them to fill a short survey about the perceived usability of the two approaches. We also conducted 5-minute post-study interviews with participants.

To determine the effectiveness of both code generators, we measure the time that participants need to complete each task. To measure preference for one approach over the other, we employ the System Usability Scale (SUS) [Bro96] and Net Promoter Score (NPS) [Rei03]. The former determines usability, while the latter measures user satisfaction with a system. Both scales transform answers to a questionnaire given by users of a system into a single number.

In SUS, a system receives a score between 0 and 100 such that higher values indicate higher usability. Tools that surpass 68 are seen as usable [Bro96]. NPS may range from -100 to +100. Systems that score below 0 are considered unsatisfactory, while results above 50 are viewed as having excellent satisfaction [Rei03]. By means of the post-study survey, we collected more direct feedback and suggestions for improvements for both CogniCrypt$_{\text{GEN}}$ and CogniCrypt$_{\text{OLD-GEN}}$.

Results

All 16 participants successfully completed both tasks in the given time window. On average, the encryption task was completed 38% more slowly with CogniCrypt$_{\text{GEN}}$ than with Cog-

niCrypt$_{OLD\text{-}GEN}$. In contrast, participants were 63.2% faster to complete the hashing task using CogniCrypt$_{GEN}$ than CogniCrypt$_{OLD\text{-}GEN}$. Investigating the overall completion times, we found no statistical significance with a Wilcoxon signed-rank test for paired data ($p > 0.05$). Initially, the mixed results came as a surprise to us. However, after evaluating the post-study interviews, we are able to attribute them to the steep learning curve for CogniCrypt$_{GEN}$. Seven out of 16 participants mentioned this issue unprompted, all of whom explained that they had not remembered all details from the introduction for either tool. However, since CogniCrypt$_{OLD\text{-}GEN}$ requires more hard-coding, they managed to get faster into implementing the requested changes. For CogniCrypt$_{GEN}$ on the other hand, they had to re-read the respective existing code to remember the underlying concepts. All seven participants mentioned that a written example-driven documentation that covered what was discussed in the introduction would likely solve this issue.

In terms of usability, CogniCrypt$_{GEN}$ fares significantly better (SUS: 76.3 and NPS: 56.3) compared to CogniCrypt$_{OLD\text{-}GEN}$ (SUS: 50.8 and NS: -43.7). Applying a Wilcoxon signed-rank test, we found the differences between CogniCrypt$_{GEN}$ and CogniCrypt$_{OLD\text{-}GEN}$ in both SUS and NPS to be statistically significant ($p = 0.005$). Overall, participants appreciated the purpose and design of CogniCrypt$_{GEN}$. In particular, participants enjoyed being able to develop code templates in Java and the structural clarity of CrySL. In the post-study interview, all but one participant preferred CogniCrypt$_{GEN}$ over CogniCrypt$_{OLD\text{-}GEN}$. This preference further reflects how significantly more usable CogniCrypt$_{GEN}$ is compared to CogniCrypt$_{OLD\text{-}GEN}$.

Although participants generally enjoyed using CogniCrypt$_{GEN}$, there is still some room for improvement—something that is underlined by the post-study interviews. Participants proposed several enhancements to the CogniCrypt$_{GEN}$ API (e.g., abandoning the call-chain design of fluent APIs, shortening the API method names, and providing content assist for class names in the `includeClass()` call). Three participants also suggested to give the code template a different name from the generated class and to add a comment to `templateUsage()` that indicates it was generated.

7.3.5 Discussion

Putting everything together, CogniCrypt$_{GEN}$ proves to fulfil the goals we set in the design phase. With CogniCrypt$_{GEN}$, we were capable of implementing eleven common cryptographic use cases, all of which are more compact than with CogniCrypt$_{OLD\text{-}GEN}$. Our experiments have shown that CogniCrypt$_{GEN}$ does not take longer than ten seconds for any of the eleven use cases with negligible memory overhead. Our user-study participants appreciated CogniCrypt$_{GEN}$ significantly more than CogniCrypt$_{OLD\text{-}GEN}$, but requested more proper documentation and made several suggestions to further improve the tool's usability.

7.3.6 Threats To Validity

Our experiments exhibit threats to validity. Since we selected graduate students as study participants, the internal validity of the study is threatened as students are not necessarily cryptography experts. Therefore, they may take longer for a given task, because they lack the knowledge for a particular cryptographic API and not as a result of the code generator. We mitigated this threat in two ways. First, the task descriptions included the cryptographic APIs and methods that participants were supposed to use. Participants were only asked to implement this solution into the given code generator. Second, we also asked participants to rate their cryptography experience on a 1–10 scale. On average, participants rated themselves at 5.2, with the median self-ascribed experience level of 5. While self-ratings come with their own

caveats [Nor97, Mah16, ASLRD99, HCP+95], we did not find statistically significant differences between participants who rated themselves higher than average with those lower than average.

The unequal familiarity of participants with CrySL and XSL threatens the internal validity of the study results as well. When asked to self-rate their experience with each on a 1–10 scale, CrySL scored an average of 5.2, while XSL only reached 1.3. To mitigate this threat, we (1) gave each participant an introduction to both code generators in the very beginning of the study (as participants had to self-assess their familiarity with XSL and CrySL at the end of the study, we asked for the familarity they had before our introduction) and (2) designed the study tasks such that the involvement of CrySL was minimal and modifications to CrySL could be made by anyone who had followed the introduction part of the study. Indeed, we found no significant correlation between the knowledge of CrySL and liking CogniCrypt$_{\text{GEN}}$ or being more effective and efficient with it.

In terms of external validity, the relatively low number of participants poses a threat. We addressed this threat by choosing participants from diverse backgrounds regarding experience with Java, Eclipse, and cryptography. We have, in addition, chosen statistical tests appropriate for the number of participants that indeed showed statistical significance.

7.4 Related Work

Apart from CogniCrypt$_{\text{OLD-GEN}}$, no previous work aims at avoiding cryptographic misuses by generating secure implementations of cryptographic use cases. However, there is indeed work that has attempted to generate either (1) API usage or (2) secure code based on specifications.

7.4.1 Generating API Usage Code

Various prior work generates usage code for APIs [BW12, MBP+15, KRZ14, KLHK10], however, they rely on mining syntactically correct usages of the respective APIs, which is not a viable approach for cryptographic APIs for reasons in Section 5.5.

7.4.2 Generating Secure Code

Code generators aiming to produce secure code amount to a huge body of research. There are, for example, code-generation approaches for implementations of cryptographic algorithms [HG14], security controllers [MM12], and security protocols [PSD04, NVLC11]. However, only two approaches address cryptographic misuse specifically. Garcia et al. [GTM14] present an Eclipse plugin Crypto-Assistant. Crypto-Assistant guides developers through the configuration of a database encryption by means of a GUI. The tool hooks into an existing plugin that provides support for configuring relational databases. When a user opts in the GUI for something to be encrypted, Crypto-Assistant generates a config file for the database-configuration plugin that (1) makes sure the data is encrypted and (2) the encryption is secure. The tool is limited to this one use case.

Kane et al. [KLCL18], on the other hand, do not devise and implement a use-case-based code generator like CogniCrypt$_{\text{GEN}}$. Instead, they implement several high-level cryptographic protocols such as Kerberos or TLS on top of existing low-level cryptographic APIs in Python. Effectively, their protocol implementations are wrapper code similar to what CogniCrypt$_{\text{GEN}}$ generates. The two approaches differ in (1) the use cases they support (high-level protocols vs. common cryptographic use cases) and (2) the way use cases may be implemented (hard-coded vs. generated through declarative specifications).

CogniCrypt$_{\text{OLD-GEN}}$ is the tool most closely related to CogniCrypt$_{\text{GEN}}$. CogniCrypt$_{\text{OLD-GEN}}$ combines an algorithm model in the variability-modelling language Clafer [JSM+19] with hard-

coded XSL templates. Using a constraint solver, CogniCrypt$_{OLD\text{-}GEN}$ fetches secure algorithms from the model. Through a wizard in the Eclipse plugin CogniCrypt, users may configure solutions for the eight supported cryptographic use cases. CogniCrypt$_{OLD\text{-}GEN}$ writes this user input as well as the selected algorithms into an XML file that, along with the corresponding XSL template, are passed to an XSL transformer that generates the Java code by filling variability points in the code template with values from the XML file. In contrast, CogniCrypt$_{GEN}$, as our experiments in Section 7.3 show, trumps CogniCrypt$_{OLD\text{-}GEN}$ in terms of usability by facilitating code templates to be in Java. The code templates are also smaller and, by construction, provably secure with respect to CrySL specifications, which is a property that hard-coded templates cannot provide. However, our study also revealed a steeper learning curve due to a fluent API to specify crypto code compared to using hard-coded XSL templates.

7.5 Conclusion

In this chapter, we have presented CogniCrypt$_{GEN}$, a code-generation tool based on CrySL. Through code templates and CrySL rules, CogniCrypt$_{GEN}$ can generate secure implementations for the most common use cases of cryptographic APIs. We have, as our evaluation shows, designed CogniCrypt$_{GEN}$ such that it executes in a few seconds regardless of the use case and can easily run on a typical developer workstation. Our evaluation also revealed low maintenance effort and generally high usability ratings from participants of our user study, especially compared to an XSL-based solution that implements similar use cases.

Future work should address the usability issues of the fluent API in CogniCrypt$_{GEN}$ revealed by the user study. Participants criticised that class names that are passed as parameters to API calls have to be specified using strings instead of, for example, enumerations. They have also suggested to use shorter API-method names and requested more proper documentation. We are grateful for their insights and intend to improve CogniCrypt$_{GEN}$, accordingly. Implementing more use cases of other APIs in CogniCrypt$_{GEN}$ to evaluate further and, if necessary, extend its expressiveness is another interesting line of research.

8 User Study

In this chapter, we empirically evaluate the effectiveness of CogniCrypt's prototypical implementation by conducting a controlled experiment. With this experiment, we aim to answer the question we posed in the thesis statement: Does CogniCrypt's integrative approach that relieves the application developer from having to know how to use Crypto APIs effectively address cryptographic misuse? To gain more specific insights, we design the study such that we can address the following for research questions:

RQ16 *Does* CogniCrypt *impact the functional correctness of cryptography application code?* We are interested to see if participants who use CogniCrypt end up producing more functional code for a given task. We measure functionality by manually assessing participants' code for functionality based on a functionality score sheet we developed.

RQ17 *Does* CogniCrypt *impact the security of cryptography application code?* Given the main claims of CogniCrypt, we are interested to see if participants who use CogniCrypt do end up producing more secure code. We measure security in terms of how many cryptographic API misuses they make.

RQ18 *Does* CogniCrypt *impact the time taken to write cryptography application code?* Given its task-based nature, CogniCrypt is supposed to save the time needed to research and understand the various details of cryptography APIs. We are interested to see if this indeed holds where participants using CogniCrypt end up finishing the cryptography tasks faster.

RQ19 *Do developers perceive* CogniCrypt *to be more usable than plain Eclipse?* Since the usability of any tool impacts its long-term adoption, we are also interested to evaluate participants' perception of CogniCrypt. We measure usability through NPS [Rei03] and direct written feedback by participants.

RQ20 *What obstacles do developers still face with* CogniCrypt*?* When users face roadblocks while using a tool, they might stop using it, even if they consider it otherwise usable. From their written feedback, we therefore also derive obstacles participants still face.

8.1 Related Work

Our experiments expand on previous empirical research on the effectiveness of countermeasures against software insecurity. Prior work has investigated several aspects relating to software

security: the role of a security-focused development process [TTCL18, AC18], the content, length, and desired structure of security warnings [GIW+], the role of resources for security knowledge [FBX+17, ASW+17, ABF+16] as well as the effect of explicitly requesting developers towards writing secure code (i.e., priming) on its security [NDG+19, NDTS18, NDT+17].

Our study most closely resembles, in design and goal, that of Nguyen et al. [NWA+17]. The main difference lies in the object of evaluation. CogniCrypt combines a static misuse detector with a code generation, supports Java, and is integrated into Eclipse, while Fixdroid is only equipped with an analysis, is limited to Android, and integrates with Android Studio. Fixdroid further only checks for a few hard-coded misuses, whereas CogniCrypt's analysis component may be parametrized by usage specifications in CrySL. However, in contrast to CogniCrypt, Fixdroid supports IDE-integrated fixes that are selectable by users.

8.2 Experimental Design

In this section, we describe the controlled experiment we designed to address this study's goal.

8.2.1 Object of the Experiment and Methodology

To measure CogniCrypt's effectiveness and answer our five research questions, we compare the cryptography code software developers write with and without CogniCrypt. To this end, we designed the experiment such that each participant is asked to implement two programming tasks that involve cryptography. For one of them, they are allowed to use CogniCrypt, for the other one they use a regular Eclipse. We compare against a regular Eclipse to most closely resemble an everyday working environment of application developers. In the following, we will refer to the environments as "CC" (for **C**ogni**C**rypt) and "EC" (for **EC**lipse).

We follow a *within-subjects* design to ensure that we can observe the effect of CogniCrypt per participant and avoid possible biases or population differences caused by the distribution of participants among two separate groups [CGK12]. A within-subjects design allows us to run the experiment with a smaller number of participants than would have been needed for a between-subjects design. It is also resilient towards variability in individual skill level since it compares scores of one participant in one condition with the scores of the same participant in a different condition. This design further provides a better chance of observing any statistical differences between the two tested environments EC and CC. To avoid learning/practice effects as well as fatigue effects that might influence the solutions, we follow a latin-square design [Gao05] where the order of the tasks and environments presented to the participants is assigned in a way such that each task appears in each sequential position an equal number of times. In other words, an equal number of participants receive each possible ordering of tasks and environments.

Before each task, we ask participants to read through a tutorial consisting of a handful of lines of text and some screenshots on the environment they would be using in the next task. The experiment instructor asks them to make use of the features mentioned and explained in the tutorial as much as possible while working on the task. We have, however, avoided to provide any particular in-depth documentation on $\text{CogniCrypt}_{\text{GEN}}$ and $\text{CogniCrypt}_{\text{SAST}}$. This decision—if anything— puts CogniCrypt at a disadvantage as participants are more likely to be familiar with regular Eclipse than with CogniCrypt and every tool comes with a learning curve. However, we did not want to unnecessarily bias participants and assumed that the more documentation we would provide the clearer it would be which of the two tools was ours. Such bias would severely limit the value of the feedback participants give us on CogniCrypt's feedback in the survey.

While solving the tasks, participants are allowed to use any online resources they want to,

Table 8.1: Tasks for Participants

Name	Goal	Program Stub
FE	Encrypt a file	Reads file and writes it back to disk
TLS	Send specific message to a server via TLS connection	Message that should be sent is defined

apart from email and chat applications. We also prime participants by enhancing task descriptions with requests to participants to pay extra attention to security while implementing the task. We do so to account for previous research that strongly suggests that developers, in the context of user studies, do not bother with security concerns unless explicitly requested [NDG+19].

We design the two tasks shown in Table 8.1. For the tasks, we implemented two small Java program stubs (involving 1–3 classes) that participants had to modify during the experiment. For each task, the participant needs to add certain security functionality to the existing program stub. Task FE requires the participant to implement a secure file encryption using a password. The program stub we provided reads the file into a string and then stores that string into a file again. In task TLS, we expect participants to implement a TLS client whose server runs locally on their machine at port 9999. For this task, the stub defines the message that should be sent. It also contains a key-store file that stores the certificate the TLS connections must use when connecting to the server. The task description pointed participants to this file.

With each stub, we provide several unit tests, each covering one requirement for functional correctness. The task descriptions explain that a task is completed once all unit tests pass. They also point participants to the exact method stubs to implement, such that they can run the unit test before submitting their code. We have further enhanced the program stubs with todo-comments at the program locations that require participants' extensions.

In summary, this study design leaves us with four different conditions. Condition 1 has the participant start with task FE using the regular Eclipse and then go on to implementing task TLS with CogniCrypt (FE/EC –> TLS/CC). In condition 2, the order is swapped (TLS/CC –> FE/EC). For condition 3, a participant first works on task TLS in regular Eclipse and subsequently continues with task FE in CogniCrypt (TLS/EC –> FE/CC). Condition 4 once again switches the order of configurations from condition 3.

8.2.2 Participants and Experiment Context

We recruited 32 graduate students at two universities to participate in the experiment. All students were either currently taking a course including Java development tasks or had completed such a course already, e.g., a course for which they had implemented several static program analyses in Java. We considered this experience sufficient in terms of Java programming skills. We did not filter based on students' knowledge of cryptography. In fact, we did not mention cryptography during recruiting to not bias our sample set towards students who have more experience with cryptography.

8.2.3 Collected Measurements

To answer **RQ16** and **RQ17**, we have compiled a score sheet of requirements that the the implementation of each task needs to exhibit in order to count as *functionally correct* or *secure*, respectively. Table 8.2 shows the criteria for both tasks. The test cases we enhanced each stub with also covered requirements for functional correctness that we were able to cover through a unit test case. We mark the requirements that correspond to a test case in a stub by '(test)'

Table 8.2: Functionality and Secuity Requirements for Study Tasks

Name	Functionality	Security
FE	Write of ciphertext file was successful (test) Ciphertext file existence (test) Ciphertext file is not empty (test) Ciphertext file is not equal to plaintext file (test) Password used for key generation Using some kind of encryption Encrypting the whole plaintext	Using secure encryption configuration Using secure key deriviation Password has never been a String Using secure hashing algorithm Random salt of at least 16 byte Secure preparation of encryption
TLS	Correct message (test) 4x Incorrect Message (test) Using provided parameters Setting correct key store Flushing of write channel Closing Connection	Using TLS Using secure SSL socket factory provider Using secure cipher suites Using secure tls protocols

in Table 8.2. We measure correctness and security of each participant by first running the test cases on their code and subsequently manually checking the compliance with the remaining functionality as well as the security criteria. The percentage of items covered are the *functionality* and *security score* of each task, respectively.

For an implementation of the FE task to be considered correct, a ciphertext file must exist that is different from the plaintext file, but not empty. The password provided through the stub must also be used to generate a cryptographic key. Finally, some form of encryption must be used—even if it is a self-implemented one—that encrypts the whole plaintext. Security-wise, the encryption configuration must be secure. That is, no insecure algorithms (e.g., DES) or block modes (e.g., ECB) must be used. The key must be derived securely from the password. That requires (1) the password to be used, (2) the key derivation to be conducted through `PBEKeySpec`, and (3) the `PBEKeySpec` to be used securely (Section 1.1). In addition, the encryption must be prepared securely. That is, depending on the cipher mode the participant uses, they may need to provide an IV.

To implement task TLS correctly, the client must be able to send a message and receive the server's answer. When it sends the correct message, it should also handle the appropriate response from the server. The client must use the correct IP and port, set the correct key store, flush the write channel, and close the connection at the end. From a security perspective, we require the implementation to actually use TLS. The TLS connection must also be set up using an appropriate socket, e.g., through the JSSE. Lastly, the TLS connection must be configured to use secure cipher suites and only enable secure TLS protocols. A default configuration of a TLS connection set up through the JSSE allows both insecure cipher suites and TLS protocols (Section 2.1.3). Participants therefore have to configure these themselves. Participants who cannot use CogniCrypt for this task therefore have to not only discover on their own that the default configuration is insecure, they also have to figure out how to enable secure cipher suites and TLS protocols only.

To answer **RQ18**, we also measure the time participants take to complete the task. We consider completion time as the time from when a participant starts to read the task description until they close the development environment. We intentionally include any time spent outside the IDE looking at online resources as we believe this is part of the time taken to complete the

task. In other words, we do not pause the timer if the IDE loses focus.

8.2.4 Survey Questionnaire

To answer **RQ19** and **RQ20**, we want to understand the steps developers take to solve a task. However, to ensure a natural work setting and to avoid inaccuracies in measuring the time a participant takes to "think aloud" approach, we do not follow that design. Instead, we ask participants to fill out questionnaires after each task. In these questionnaires, we ask about the perceived difficulty of the task, the clarity of the task description, and their experience with the environment. For each task, the questionnaire includes the following questions:

Q1/8: This task was difficult.

- Type: Scale 1 (Strongly Disagree) to 5 (Strongly Agree).

Q2/9: This task was fun.

- Type: Scale 1 (Strongly Disagree) to 5 (Strongly Agree).

Q3/10: I think I solved this task correctly.

- Type: Scale 1 (Strongly Disagree) to 5 (Strongly Agree).

Q4/11: I think I solved this task securely.

- Type: Scale 1 (Strongly Disagree) to 5 (Strongly Agree).

Q5/12: Have you written or seen code for tasks similar to this one before? For example, maybe you worked on a project that included a similar task, but someone else wrote that portion code.

- Type: Single-answer question.
- Possible choices: [I have written similar code; I have seen similar code, but have not written it myself; No, neither; I don't know; I prefer not to say.]

Q6/13: Based on the last task, please how likely you would recommend the coding environment to a fellow programmer.

- Type: Scale 0 to 10.

Q7/14: Please provide additional feedback about what you found useful or not useful in the coding environment. Please provide concrete examples when possible.

- Type: Free-text field.

We also provide another questionnaire at the end of the experiment (i.e., after finishing both tasks) that asks participants about their programming experience **Q15** –**Q20**, security expertise **Q21** – **Q26**, general demographics **Q27** – **Q31**, and more general comments on their experience **Q32**. This final questionnaire includes the following questions:

Q15: How many years have you been programming in Java?

- Type: Free-text field.

Q16: How many years have you been coding in general?

- Type: Free-text field.

Q17: How did you learn to code?

- Type: Multiple-answers question.
- Possible choices: [Self-taught, Online class, College, On-the-job training, coding, other]

Q18: Which coding environment do you primarily use (name of the IDE or text editor)?

- Type: Free-text field.

Q19: Are you currently using the Eclipse IDE?

- Type: Single-answers question.
- Possible choices: [Yes, No]

Q20: How long have you been using the Eclipse IDE (in years)?

- Type: Free-text field.

Q21: Do you have an IT-security background?

- Type: Single-answers question.
- Possible choices: [Yes, No, Prefer not to say]

Q22: If you answered yes in the previous question, please specify.

- Type: Free-text field.

Q23: Which of the following options do describe your experience with cryptography best?

- Type: Single-answers question.
- Possible choices: [I have never used cryptography during software devlopment, I invented my own cryptographic algorithm or protocol, I implemented a cryptographic algorithm or protocol, I occasionally use cryptographic APIs or libraries during software development, I don't know, Prefer not to say]

Q24: Have you taken a computer-security class or course in the last five years?

- Type: Single-answers question.
- Possible choices: [Yes, No, Prefer not to say]

Q25: If you answerd yes in the previous question, please specify.

- Type: Free-text field.

Q26: Please tell us your highest degree of education.

- Single-answer question.
- Possible choices: [Less than high school, High school, Some college, Bachelor's degree, Master's degree, Professional degree, Ph.D., Prefer not to say.]

Q27: Please tell us your gender. ("na" for "prefer not to say")

- Type: Free-text field.

Q28: How old are you?

– Type: Free-text field.

Q29: What country to you live in?

– Type: Free-text field.

Q30: What is your native language?

– Type: Free-text field.

Q31: What other languages are you fluent in (if any)?

– Type: Free-text field.

Q32: Please provide any additional feedback for the study you have.

– Type: Free-text field.

All three questionnaires were created using Google Forms.

8.2.5 Pre-Testing

We first conducted a pilot study with five test participants. For the purpose of the pilot study, we followed the same study design as described above, apart from one aspect: We aimed at receiving feedback from participants of the pilot study to refine our study design if necessary and were not attempting to take exact time measures. As a result, we *did* follow the "think aloud" approach for the pilot study.

Participants informed us of ambiguities in the task descriptions and confusing oddities in some UI elements of CogniCrypt. For the final study, we revised the formulations in questions and re-designed the respective UI elements. All pilot-study participants finished both tasks, including questionnaires, within 45 to 60 minutes. In the final experiment, we hence told participants to finish within an hour. After half the time, we reminded them to move on to the second task if they had not already. From the results we report in this thesis, none have been gained from the pilot study.

8.3 Results

From originally 32 participants, we had to exclude the results of eight because they neither executed $\text{CogniCrypt}_{\text{GEN}}$ nor $\text{CogniCrypt}_{\text{SAST}}$ throughout the whole study. For the remaining 24 (*P01 – P24*), we show the participant distribution among the conditions in Table 8.3. We only manually analyzed of the participants' code and their survey answers for the remaining 24 participants, too. The manual analysis was conducted by the author of this thesis and a co-author of the research paper this chapter is based upon. The agreement ratio for functionality and security score are 92% and 90%, respectively. For all differences in rating, the two raters negotiated until they reached a compromise. Table 8.4 provides a complete overview of all results of the 24 participants.

8.3.1 Functionality (RQ16)

For each solution, we first investigate whether it actually implemented the task completely. To this end, we first run the test cases. In the following, we will distinguish between running and broken solutions. For us to consider a solution *running*, all the provided unit test cases must terminate without exception, even if they fail. We consider a solution *broken*, on the other

Table 8.3: Conditions & Participants

Condition	Particpants
EC/FE -> CC/TLS	7
CC/TLS -> EC/FE	4
EC/TLS -> CC/FE	7
CC/FE -> EC/TLS	6

hand, when at least one of its test cases throws an exception. We make this distinction because non-running programs are distinctly non-functional in comparison to a program that does not implement all functionality, but at least terminates. To appropriately account for this, we award all non-running programs zero functionality points regardless of the state of their implementation and discard them from the remainder of this discussion.

Our results indicate that there is a noticeable difference between the running/broken ratios of solutions that have been implemented using CogniCrypt and those without. Without CogniCrypt, participants only produced running code for the FE task in six out of eleven cases. For task TLS, only two participants managed to get the test cases working. Two further participants succeeded at establishing a connection, but failed at sending data, causing the test cases to hang. Everyone else but two participants did not manage to establish a connection to begin with. In contrast, *with* CogniCrypt, participants produced running code for all but one case for both task FE (twelve out of thirteen) and task TLS (ten out of eleven).

Participant *P07* who did not manage to complete task FE with CogniCrypt did use CogniCrypt$_{\text{GEN}}$ and had it generate the `templateUsage()` method into the correct class. However, they subsequently ignored the generated code, attempted to implement a custom solution that throws an exception when the test cases are run. Participant *P15* failed to implement task TLS for a similar reason. They also used CogniCrypt$_{\text{GEN}}$ to generate code, but for the encryption use case. They subsequently tried to manually set up the TLS connection and used the generated `templateUsage()` only to encrypt the message. As their code does not compile, the test cases cannot be executed.

As mentioned above, we will limit the following discussion to solutions that can be run. Figure 8.1 shows the functionality scores across all four combinations of environment/programming task. The score is shown in percentages, that is, a score of 2 out 4 is shown as 0.50. In our following discussion, when we justify scorings, we refer to individual points instead of the percentages.

FE For task FE, three of the six participants who completed the task *without* CogniCrypt achieve a full functionality score, resulting in a mean functional score of 1. We deducted one point from the other three solutions because they all failed to derive the encryption key from the password. From the twelve participants who implemented task FE successfully *with* CogniCrypt, eight did so with a full functionality score. Participant *P23* generated code for the wrong use case ("Secure Password Storage") and then attempted to use it in a custom-made encryption. The remaining three participants did manage to run CogniCrypt$_{\text{GEN}}$, but then failed to integrate the generated code in one way or another. *P02* completed the implementation of the encryption, but did not manage to store the result of the encryption in the variable the stub writes into the ciphertext file. Hence, the ciphertext file has the same content as the plaintext, although the encryption itself was implemented in a functionally correct manner. *P01* ignored the generated code. They even went so far as to delete the method `templateUsage()` from the Java file they coded in, but left the other generated code untouched. Instead, they

Table 8.4: Participants Overview

Participant	Condition		Score Task 1			Score Task 2		
	Task 1	Task 2	CogniCrypt	Functionality	Security	CogniCrypt	Functionality	Security
P01	TLS	FE	○	0/9	0/4	●	4/7	0/6
P02	FE	TLS	●	5/7	5/6	○	0/9	0/4
P03	FE	TLS	○	6/7	1/6	●	9/9	2/4
P04	TLS	FE	○	0/9	0/4	●	7/7	6/6
P05	FE	TLS	●	7/7	6/6	○	0/9	0/4
P06	FE	TLS	○	6/7	1/6	●	9/9	4/4
P07	TLS	FE	○	0/9	0/4	●	0/7	0/6
P08	FE	TLS	○	7/7	6/6	●	9/9	4/4
P09	TLS	FE	○	0/9	0/4	●	6/7	6/6
P10	FE	TLS	○	0/7	0/6	●	9/9	4/4
P11	TLS	FE	●	9/9	4/4	○	0/7	0/6
P12	FE	TLS	●	7/7	6/6	○	0/9	0/4
P13	FE	TLS	●	7/7	6/6	○	0/9	0/4
P14	FE	TLS	○	7/7	1/6	●	9/9	4/4
P15	TLS	FE	●	0/9	0/4	○	0/7	0/6
P16	TLS	FE	○	0/9	0/4	●	7/7	5/6
P17	TLS	FE	●	9/9	4/4	○	6/7	1/6
P18	TLS	FE	●	9/9	4/4	○	0/7	0/6
P19	FE	TLS	○	7/7	4/6	●	9/9	4/4
P20	TLS	FE	○	0/9	0/4	●	7/7	6/6
P21	FE	TLS	●	7/7	6/6	○	3/9	2/4
P22	FE	TLS	○	0/7	0/6	●	9/9	4/4
P23	TLS	FE	○	2/9	2/4	●	7/7	6/6
P24	FE	TLS	●	4/7	1/6	○	0/9	0/4

● indicates that the task was performed with CogniCrypt and ○ without, respectively.

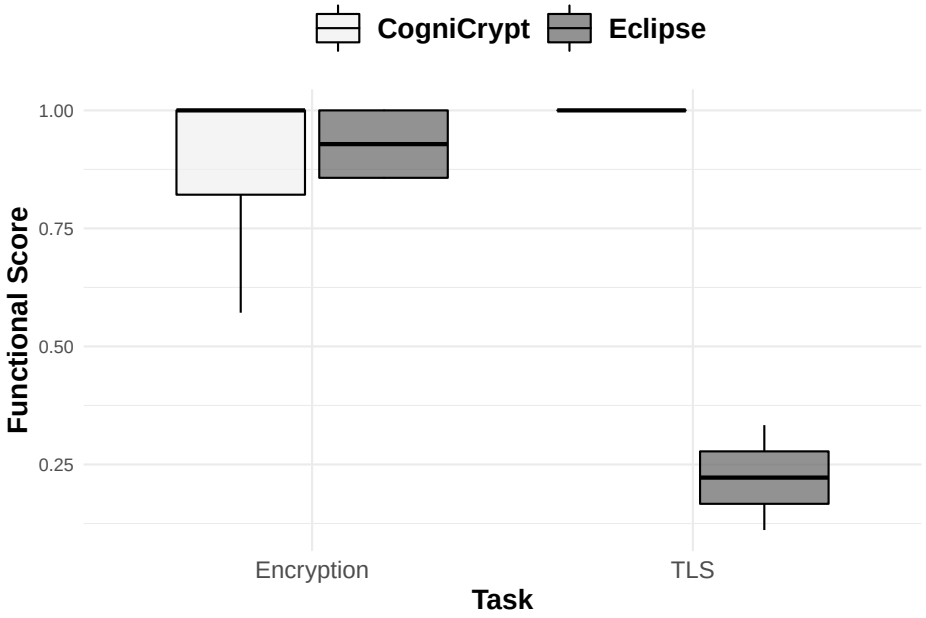

Figure 8.1: Box Plot Showing the Functionality Score for Tasks FE and TLS in Environments CogniCrypt and Regular Eclipse.

implemented a custom solution that they did not manage to finish. Lastly, *P09* only made use of the key-generation code and attempted to implement a custom encryption solution for the data using `CipherOutputStream`. However, this solution does not encrypt the whole plaintext.

TLS For task TLS, the results are much clearer than for task FE. First, as mentioned above, only two participants who did not use CogniCrypt for the task, did actually produce running code. Those two received two (*P23*) and three (*P21*) out of nine points on the functionality score, respectively. In neither submission do any of the test cases pass nor do they set the correct keystore. *P23*, in addition, fails to flush the channel to the server. In contrast to that, all ten participants who implemented task TLS using CogniCrypt received full points on the functionality score. They all generated code using $\text{CogniCrypt}_{\text{GEN}}$ and integrated it properly into the program stub.

Summary In conclusion, for the eight participants who implemented task FE with CogniCrypt and received full points on the functionality score, CogniCrypt worked as intended. Some participants did, however, face problems. In one case, the participant was not clear about which use case they need to pick. For the remaining three, CogniCrypt failed at properly communicating that and how they need to integrate the generated code into their own application code. On the other hand, all but two participants attempting to implement task TLS without CogniCrypt failed to so, while all but one who did use CogniCrypt succeeded. For both tasks, we found participants achieved statistically significant better scores using a Wilcoxon signed-rank test for paired data ($p < 0.05$).

We have also checked for correlations of functionality score and self-reported experience in programming (Q15–Q17), Eclipse (Q18–Q20), security or cryptography (Q21–Q25) or general demographics (Q26–Q32), but were not able to find any. Similarly, we have not found any correlations between functionality score and order of task or tool.

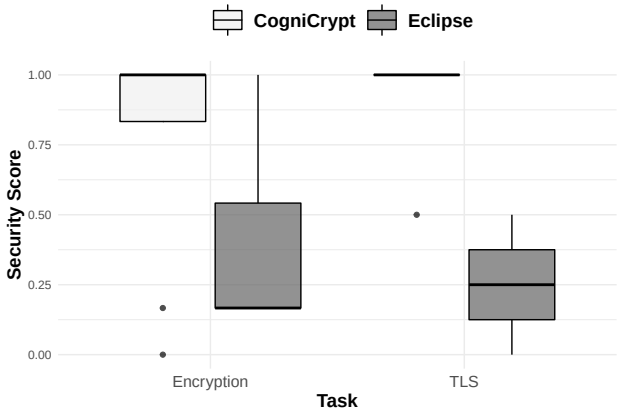

Figure 8.2: Box Plot Showing the Security Score for Environments CogniCrypt and Regular Eclipse and Tasks FE and TLS.

8.3.2 Security (RQ17)

We also observe similar trends for the security score. We show the distribution over the four environment/task combinations in the box plot in Figure 8.2.

FE For task FE, similar to the functional score, we find again somewhat ambiguous results, although much less so than for the functional score. Only one of the six participants who implemented task FE *without* CogniCrypt achieved a full security score. Participant *P19* achieved four out of six security points, only lacking a random salt and choosing an insecure iteration count. The remaining four participants all received one out of six points. From the twelve participants who *did use* CogniCrypt, eight received a perfect score. We removed one point each for participants *P02* and *P16* because they transformed the password from a `char` array into a `String`. Neither of the two ended up using the String password variable, making it effectively dead code and likely to be optimized away by the Java compiler. We also assume this code would have been cleaned up in any real-world setting, but decided to remove the point nonetheless because the code as-is is insecure. *P01* and *P23*'s custom solutions, which we already discussed above, do not hold any security guarantees. While, for instance, *P01* generates a cryptographic key from the password (which is why they get a point on the functional score for this requirement), they do so using a number of String, hashing and array-copy operations. The code also transforms the password into a `String`. In total, *P01*'s solution receives zero points.

TLS Participants who implemented task TLS *with* CogniCrypt all received a perfect security score. This is because the code generated through CogniCrypt only enables secure cipher suites and TLS protocols. The two participants who implemented at least a *running* program for task TLS without CogniCrypt achieved zero (*P023*) and two points (*P021*), respectively. We removed two points for participant *P021* because they neglected to configure the connection in terms of cipher suites and TLS protocols.

We also checked the security of the broken solutions for task TLS developed without CogniCrypt to provide at least some kind of evaluation. None of the ten participants would receive more than two points because they display the same problem as *P21*'s solution. For the reasons we explained above, we do not include this data into the box plot, however.

Summary In summary, participants fare better in terms of security for both tasks when using CogniCrypt, compared to when they try it without. As with the functionality score, we were able to show a statistically significant improvement with CogniCrypt, through a Wilcoxon signed-rank test for paired data ($p < 0.05$). For participants using CogniCrypt, the only points detracted were for failing to clean up the code and for complete custom-made solutions. We have again checked for correlations with any of the forms of experience we surveyed participants about in the questionnaire as well as the order of tasks and tools, but could not find any.

8.3.3 Completion Time (RQ18)

We report the distribution of completion time in Table 8.3. As with functional and security score, we only report completion times for non-broken solutions. We first note that completion times for participants using CogniCrypt spread comparatively widely. For task TLS, *P18* finished in six minutes and thirty-two seconds, whereas it took *P11* about 39 minutes and 30 seconds. The fastest successful participant for task FE completed their work in not even two minutes. In stark contrast, the participant who took the longest needed about 42 minutes. We attribute this wide range to two behaviours we observed while conducting the study when walking around the room and watching participants over the shoulder. Many participants, when they had CogniCrypt available, first attempted to finish the task without using either $\text{CogniCrypt}_{\text{GEN}}$ or $\text{CogniCrypt}_{\text{SAST}}$. Most eventually gave up, resorting to either launching $\text{CogniCrypt}_{\text{GEN}}$'s wizard or triggering $\text{CogniCrypt}_{\text{SAST}}$. Second, some participants took longer than others to generate code for the correct solution using CogniCrypt. Some appeared to struggle when having to answer questions in $\text{CogniCrypt}_{\text{GEN}}$'s wizard. Others even generated code for an incorrect use case at first.

FE When comparing the two plots for completion time with task FE directly, participants generally seem faster with CogniCrypt. However, the slowest participant *with* CogniCrypt seems slower by several minutes than the slowest participant *without* CogniCrypt. The diagram presents a somewhat skewed picture, however, because more participants managed to finish when *using* CogniCrypt compared to when *not* using it (twelve of thirteen vs. six out of eleven). We find it likely that participants who took longer *with* CogniCrypt would not have finished if they had not had it at their disposal.

TLS For task TLS, the median completion time lies at around fourteen minutes when using CogniCrypt. The two participants who finished TLS without CogniCrypt completed their work faster. However, as both the functional and security scores of the two indicate, their solutions are far from being actually complete and secure. On top of that, we also argue again that many participants who took longer with CogniCrypt would not have produce *running* code without it. The high number of participants who did not produce running code without CogniCrypt serves as a strong indicator for this claim. Given these two observations, we conclude that CogniCrypt improves the completion time for this task.

Summary We conclude that participants are significantly faster with CogniCrypt. We come to this conclusion because of (a) the higher completion rates in a setting with limited time available and (b) the lower median completion times for the solutions that were completed. Where CogniCrypt seems to be slower than regular Eclipse, we assume the slower speed to be more indicative of more people being enabled to finish a task to begin with.

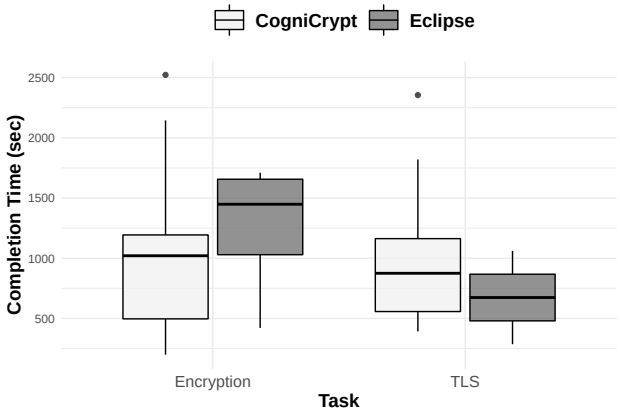

Figure 8.3: Box Plot Showing the Completion Time for Environments CogniCrypt and Regular Eclipse and Tasks FE and TLS.

8.3.4 Usability (RQ19)

Participants generally expressed positive views on CogniCrypt's usability, but provided criticism relating to the integration with Eclipse of both $\text{CogniCrypt}_{\text{GEN}}$ and $\text{CogniCrypt}_{\text{SAST}}$. In contrast, regular Eclipse overall received substantially worse reviews by participants. These results are reflected in the both tools' NPS values. As explained in Section 7.3.4, NPS generally measures user satisfaction and ranges from -100 to 100, whereas any value above 50 is considered excellent and values below 0 are considered bad [Rei03]. CogniCrypt receives an NPS value of 33.33, a result that is generally considered good, but not excellent [Rei03]. Regular Eclipse, on the other hand, receives an NPS of -54.17.

When asked for more concrete feedback (Q7/Q14), participants especially praised $\text{CogniCrypt}_{\text{GEN}}$ (*P02 – P07*, *P09*, *P11*, *P12*, *P16*, *P19*, *P21*). *P02* notes that they "*have never worked on encryption before in Java.*" However, they would "*strongly say that [*CogniCrypt *] will be very helpful with prior knowledge [of the tool].*" *P05* adds they found $\text{CogniCrypt}_{\text{GEN}}$ useful and that it was "*easy to choose input and generate the encryption function by the tool. It was very user friendly for developer since there was no further need to read about the encryption function and security/authenticity of that encryption library*". Both aspects highlighted by *P05*—the high effectiveness for developers with low experience and that CogniCrypt lifts a burden off a developer because it takes care of security features—are also shared by other participants. *P19*, for instance, finds $\text{CogniCrypt}_{\text{GEN}}$ "very helpful" because they could "*assume[...] that [the] generated code [was] secure, which saved a lot of time searching for documentation on security requirements*". *UPB18* also highlights the usefulness for cryptography novices: "*Since I have never worked on anything similar before, the code generator ([for task TLS]) was very helpful for me to get an idea about what I have to do. It was easy to use, since every single configuration was asked.*" *P12* agrees as they "*haven't worked that much with encryption so far, which is why the code generation was especially helpful.*" *P04* highlights the effort that is saved through $\text{CogniCrypt}_{\text{GEN}}$, because they do not "*have to search the web for the right and required classes to get the job done.*" *P03* goes so far as considering code generators like $\text{CogniCrypt}_{\text{GEN}}$ necessary for "*secure software engineering*", because "*we can be sure our code is almost secure.*"

P06 appreciates $\text{CogniCrypt}_{\text{SAST}}$ as helpful because it detected their use of an inappropriate block cipher mode for the call `Cipher.getInstance("AES")`. *P13* tested $\text{CogniCrypt}_{\text{SAST}}$ by purposefully introducing misuses to the code and enjoyed that it found them.

P23 praised CogniCrypt to be "*well integrated*" and *P20* found it generally "*easier to use*".

8.3.5 Obstacles (RQ20)

One obstacle several participants clearly faced was the integration of generated code into their project. This obstacle is, on the one hand, demonstrated by the three participants we described above who had CogniCrypt$_{GEN}$ generate code for task FE, but then failed to properly call that code. However, participants have also mentioned CogniCrypt's shortcomings in that respect in their feedback. *P21* comments that "*it was difficult to find where the autmomatic generated code is located*". *P01* requests the "*comments should be improved for auto-generated code*", because it took them "*a while to understand what the generated code says and to identify which part of the code is generated when [they] already have [their] own code in the class. So there should be clear way to separate automatic generated code from [their] own code.*" *P08* encountered the same problem as they needed time to understand which classes CogniCrypt$_{GEN}$ had generated for them to implement task TLS. When implementing task FE, *P13* struggled to identify, if the generated code would already derive the key from the password or if they had to implement that themselves. In the end, they did not implement it themselves, because the tests passed.

One further problem with CogniCrypt$_{GEN}$ in particular appears to be the usability of its wizard. As we noted above, several participants had to go through multiple attempts of generating code. The issue is also highlighted by *P22* who criticises that they at first "*didn't notice the first screen of the wizard provided a choice, [they instead] mistook it for an introductory page and just clicked 'continue'[, and were] then confused that the code generated didn't suit my needs*".

The warning messages by CogniCrypt$_{SAST}$ provided another major obstacle participants reported on. *P13* rightly complains that it "was confusing [...] that the autogenerated code throws errors (in [CogniCrypt$_{SAST}$]), which on further inspection are only in the decryption case and not applicable." *P02* ran into the same issue when they were "getting a[n] error at [the encryption] method at line number 67. Which [they] could not understand that what is exactly the problem." The warnings the two participants report of relate to long-known false positives in CogniCrypt$_{SAST}$ that have been addressed since conducting the study. *P07* raises a further issue with CogniCrypt$_{SAST}$'s reporting: "*But when we are using functions with more than two arguments it is only returning problem saying the parameter is not properly generated, additionally it should also return info about how to correct it or some possible description for the developer to enhance the code for better efficency.*" The warnings, both *P02* and *P07* refer to, CogniCrypt$_{SAST}$ displays for predicate violations. We have indeed been struggling to find a good wording for these kinds of violations and have been editing CogniCrypt$_{SAST}$ warning messages for such misuses upon feedback by users several times since CogniCrypt$_{SAST}$'s original publication [KSA+18]. This comment shows there is still work to be done regarding this matter.

Lastly, *P01* criticizes that CogniCrypt$_{GEN}$ and CogniCrypt$_{SAST}$ "*lack documentations, tooltips which detail the options they provide.*" This point is valid for the study, its applicability in practice, however, may be limited. As mentioned before, we intentionally kept the documentation limited to not unnecessarily bias participants. In real-world contexts, users would have the documentation available on CogniCrypt's website[1], which provides extensive introductions to all its components.

8.4 Discussion

Our controlled experiment has shown CogniCrypt to be effective. Participants were significantly faster in implementing application code that requires using cryptography concepts. The code they produced was significantly more functional *and* secure than when only using Eclipse.

[1] `www.cognicrypt.org`

Participants generally judge CogniCrypt to be a useful and usable tool as its NPS score underlines. Our results therefore allow us to conclude for **RQ16** to **RQ18** that CogniCrypt has a significant positive impact on all three. In addition, we can answer **RQ19** such that developers do indeed seem to view CogniCrypt as more usable than plain Eclipse.

However, there is still room for improvement. In response to **RQ20**, we can report two main findings. First, when it comes to integrating code by $\text{CogniCrypt}_{\text{GEN}}$ into their application, a large subset of participants struggled because they had trouble understanding what is happening in their IDE when $\text{CogniCrypt}_{\text{GEN}}$ generated code. To help with the situation, we plan to have $\text{CogniCrypt}_{\text{GEN}}$ inform users better. $\text{CogniCrypt}_{\text{GEN}}$ should enhance method `templateUsage()` with a comment describing which pieces of code have been generated, what their purposes are, and where they each can be found. A second issue raised by participants revolves around the error messages produced by $\text{CogniCrypt}_{\text{SAST}}$, in particular error messages on predicate-related misuses. We have refined predicate error messages several times, each time reducing the knowledge of CrySL and understanding of predicates in CrySL required to understand them. However, we still do not seem to be where we should be in terms of comprehensibility. Participants find the warning messages too abstract. They also criticize that they are not actionable, that is, they do not provide help as to how to resolve them. In future work, we might explore more focused usability testing for $\text{CogniCrypt}_{\text{SAST}}$'s error messages to A/B test alternative phrasings or even not showing transitive predicate warnings at all.

8.5 Threats to Validity

Our sample set poses an internal threat to the experiment's validity because it only consisted of graduate students from two universities. That sample set is likely not representative of the whole Java developer community. However, we argue that students are more likely to be less experienced in programming, especially programming in a specific language, than the average developer. We further assume grad students are only, if at all, slightly unrepresentative in terms of security knowledge. If CogniCrypt manages to support the demographic of students effectively, we expect it to fare even better with professional developers as they will likely have more experience with IDE-integrated code generators and program analysers. Recent work by Naiakshina et al. [NDGS20] supports this assumption empirically.

As far as external validity is concerned, the experiment's relatively low number of participants may pose a threat. To address this threat, we have employed the appropriate statistical means to check for correlations.

Our experiment lastly exhibits an ecological threat, too. Although we tried to come up with seemingly realistic tasks, both the task descriptions and the stubs are fairly unlikely to be found in practice exactly like that. The former rather resemble assignments in a programming course, the latter are comparatively tiny. It is further questionable as to how far the regular-Eclipse condition can be claimed to simulate an authentic environment for everyday development. Developers often configure their editor and IDEs to their liking. They also use a wide range of editors and IDEs to develop in Java and some participants had no experience in Eclipse. To mitigate that threat, we asked the participants for their experience level with Eclipse and found no statistically significant correlation with their scores or completion rates in the study. On top of that, these restrictions apply to lab studies in general and our study does not display stronger limitations than other comparative ones. As a result, we do not believe the setting albeit not necessarily representative of developers' work environment to cause the study's results to be of less significance.

8.6 Conclusion

Through this user study, we have demonstrated CogniCrypt's effectiveness. The tool significantly improves how fast, functional, and secure developers code cryptography applications. Participants' criticism where they mentioned it related mostly to implementation details that can and should be addressed, but that also in no way threaten the validity of the concept underlying CogniCrypt—or even its concrete prototypical implementation.

Our results therefore do not require us to falsify our thesis statement. The evidence we have collected in this user study heavily suggests that CogniCrypt does meaningfully combat cryptographic misuse by relieving the software developer of knowing to how use Crypto APIs.

9 Further Applications of CrySL

In Chapter 5, where we introduce CrySL, we argue CrySL rules may serve as a building block for various kinds of tool support. Chapters 6 and 7 then discuss two types of tool support that are part of the core contribution of this thesis. In this chapter, we aim to demonstrate that our claim about CrySL's expressive power and flexibility is not unfounded. To this end, we discuss more types of tool support that (could) build on top of CrySL than the two previous ones. We first present one other application CrySL has already been used for: CryptoOracle [Hol19]. CryptoOracle is a library that performs runtime checks for misuses of the JCA.

Subsequently, we also sketch designs for three not-yet-implemented tools that may be built on top of CrySL. Their design and implementation is not part of this thesis' contribution. For the purpose of this discussion, we move past the most obvious candidates such as a dynamic or hybrid program analysis. The development of either would certainly be worthwhile to patch up CogniCrypt$_{\text{SAST}}$'s shortcomings, in particular the tool's false positives. Such tools would come with their own challenges, however, as far as the integration of CrySL rules is concerned, they would work fairly similarly to CogniCrypt$_{\text{SAST}}$. We hence do not see many immediate CrySL-related conceptual challenges arising from their design and focus our discussion on other ideas. In particular, we discuss the program-repair tool CogniCrypt$_{\text{FIX}}$, the test-case generator CogniCrypt$_{\text{TEST}}$, as well as a documentation generator CogniCrypt$_{\text{DOC}}$, all of which are based on CrySL. The three approaches are in the process of being fully designed and implemented by André Sonntag, Rakshit Krishnappa Ravi, and Ritika Singh in their respective master theses.

9.1 CryptoOracle – Wrapper Library with Runtime Checks

We first discuss CryptoOracle [Hol19]. CryptoOracle is a CrySL-based library that, bootstrapped with the Ruleset$_{\text{FULL}}$ rule set, applies runtime checks to detect misuses of the API. This way, developers may test their application for misuses by simply running it. Whenever a runtime check encounters a misuse, it throws an exception.

CryptoOracle first and foremost aims to provide a direct means of support for using the JCA that is independent of external resources such as program analyzers or documentation. One further goal of CryptoOracle is to keep the overhead for application developers – the library's target audience – comparatively low. Consequently, CryptoOracle does not require developers to install a new provider or to use an entirely new API or library. Instead, the library conducts checks on a running application that uses the JCA for its compliance with Ruleset$_{\text{FULL}}$. In order to receive support from CryptoOracle, a developer must only add

the library as a dependency to their application, re-compile the program, and execute it. When the developer runs the application, CryptoOracle hooks into uses of the JCA and weaves its checks around them. To this end, the library makes use of AspectJ [KHH+01], a framework for Aspect-Oriented Programming (AOP) [KLM+97] in Java. AOP is a programming paradigm that facilitates the implementation of distinct features that are of cross-cutting concern into different modules. In practice, AOP has often been used to implement logging and other non-fuctional features. This way, logging-related code is not spread across the whole program and clogs up the code for functional features as is the case in regular object-oriented programming. In the context of CryptoOracle, Hollmann uses AOP for runtime checks for misuses of the JCA. CryptoOracle first detects when a class from the JCA is used and then weaves its correctness check around the use of that class. If it finds the class to be misused, CryptoOracle produces an exception akin to the error messages of CogniCrypt$_{SAST}$.

Hollmann [Hol19] conducted an evaluation of CryptoOracle on 46 Java projects from GitHub that use the JCA. He first compared compile times of the applications with and without CryptoOracle. He found compile-time overheads ranging from 23% up to 171%, but was unable to identify the exact causes for the overhead. Subsequently, Hollmann executed the projects and found 55% of them to contain at least one misuse. These findings provide evidence that CryptoOracle does indeed effectively support developers in finding misuses of Crypto APIs.

9.2 CogniCrypt$_{fix}$ – Fixing Cryptographic Misuses in Vulnerable Code

As the first of three hypothetical tools, we suggest CogniCrypt$_{FIX}$. CogniCrypt$_{FIX}$ would be a program-repair tool that accepts CrySL rules and a target program and fixes all violations of the former in the latter. Such a tool may support developers who struggle to fix warnings raised by CogniCrypt$_{SAST}$ (or a similar program analysis). Problems with understanding program analyzers' error messages are widespread [JSMB13] and, evidently, also occur for CogniCrypt$_{SAST}$ (**RQ20** in Section 8.4). Such problems may be mitigated through a tool like CogniCrypt$_{FIX}$ that automatically fixes the misuses its accompanying analysis tool finds.

CogniCrypt$_{FIX}$ would run in several iterations over a target program until a fixed point is reached. Either all misuses have been resolved or the tool is not able to fix any further issues. CogniCrypt$_{FIX}$ would only apply a fix for a misuse when its fix does not produce compiler errors. Otherwise, it would discard the fix. For the purpose of the following discussion, we assume CogniCrypt$_{FIX}$ to operate on source-code level. A similar program-repair system that works on byte-code level would work similarly.

We display CogniCrypt$_{FIX}$'s workflow in Figure 9.1. As a first step, CogniCrypt$_{FIX}$ would apply a program analysis tool that finds cryptographic misuses to the target program ①. For simplicity's sake, we henceforth assume this analysis to be CogniCrypt$_{SAST}$. CogniCrypt$_{SAST}$ reports the four error types we discussed in Section 6.1. When fixing misuses, CogniCrypt$_{FIX}$ would address misuse types separately one after the other. To fix violations, CogniCrypt$_{FIX}$ would maintain a list of CrySL rules involved in the violation. From these, it may fetch information such as a class' state machine, constraints, or forbidden methods.

CogniCrypt$_{FIX}$ may fix forbidden methods ② in one of two ways. If the CrySL rule specifies an alternative, CogniCrypt$_{FIX}$ may replace the forbidden call through its alternative. Parameters that are common to both the forbidden and the alternative call are merely adopted. If the alternative does not include a parameter the forbidden method comes with, CogniCrypt$_{FIX}$ would ignore it. It would, however, not delete the object from the code. In the case of the alternative needing new parameters, CogniCrypt$_{FIX}$ may take a similar approach to the one by CogniCrypt$_{GEN}$ (Section 7.1). In case no alternative call is offered, CogniCrypt$_{FIX}$

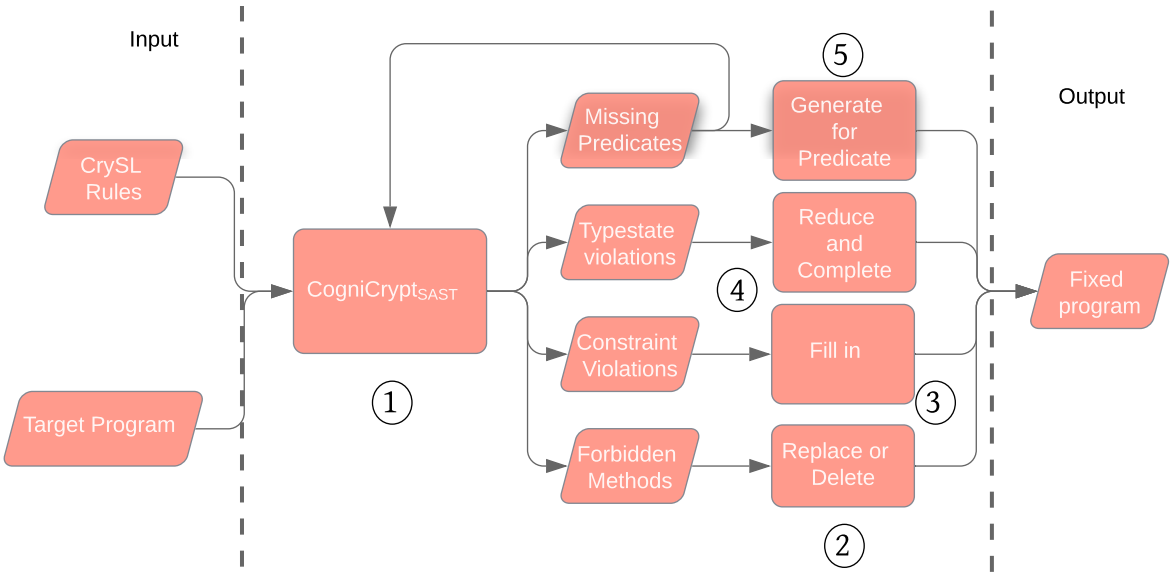

Figure 9.1: The General Workflow of CogniCrypt$_{FIX}$.

would remove the call altogether.

For parameter-constraint errors ③, CogniCrypt$_{FIX}$ would retrieve a correct value from the constraint that is violated. When choosing a correct value, it may again follow an approach similar to CogniCrypt$_{GEN}$'s for selecting parameter values (Section 7.1).

There are two types of typestate-related misuses CogniCrypt$_{FIX}$ may need to fix ④. First, an object may miss certain calls in order for it be used securely. In this case, CogniCrypt$_{FIX}$ would generate calls until the object's respective state machine reaches an accepting state, following CogniCrypt$_{GEN}$'s approach. Second, CogniCrypt$_{SAST}$ has encountered one call on an object where it expected a different one according to the corresponding CrySL rule's state machine. In this scenario, CogniCrypt$_{FIX}$ would reduce the problem to the previous case, that is, it would remove all subsequent calls on the object after the last one it expected according to the object's state machine. Following CogniCrypt$_{GEN}$'s approach, CogniCrypt$_{FIX}$ would generate appropriate calls one after another. To simplify the resolution of method parameters for the calls CogniCrypt$_{FIX}$ has to generate, it may also store the calls it deletes from the program. Equipped with this information, CogniCrypt$_{FIX}$ may retrieve the parameters for a new call, if it existed in the original source code as well.

CogniCrypt$_{FIX}$ would only fix predicate-related misuses ⑤ when all others have been addressed or abandoned. Predicate errors are often of transitive nature, that is, they often originate from misuses in other objects (e.g., a key being generated insecurely, causing the encryption that is using the key to receive a predicate error). By fixing other issues first, CogniCrypt$_{FIX}$ would avoid generating code where it is not necessary. For the predicate errors that are left, CogniCrypt$_{FIX}$ may generate code that ensures the missing predicate. As with the other misuse types, CogniCrypt$_{FIX}$ could adopt the generation approach of CogniCrypt$_{GEN}$ to do so.

After addressing all issues, CogniCrypt$_{FIX}$ would return the fixed program in Java source code. CogniCrypt$_{FIX}$ should also generate a report of which errors it did fix and which it had to leave in the code. If it were to be integrated with the Eclipse plugin CogniCrypt, CogniCrypt$_{FIX}$ could provide Eclipse quick fixes. This integration would also significantly increase the usability of CogniCrypt$_{SAST}$ within the plugin's context.

9.3 CogniCrypt$_{\text{test}}$ – Generating Test Suites for APIs

We further propose CogniCrypt$_{\text{test}}$. CogniCrypt$_{\text{test}}$ would generate test suites for an API based on the CrySL rules for this API. Such a test suite is useful from multiple perspectives. First, it provides a comprehensive suite of valid examples for how to use the API. Many APIs often lack such examples although they would provide great support for novices and, consequently, are often requested by users [NKMB16, HZH19]. Second, the test suite may serve as a test for the CrySL rules themselves. If CogniCrypt$_{\text{test}}$ generates a test case, i.e., a sample use, for an API that constitutes an invalid use of it, this test case reveals a mistake in the CrySL rule. (Or in CogniCrypt$_{\text{test}}$, of course.) Lastly, the test suite may be used to test program-analysis tools such as CogniCrypt$_{\text{sast}}$. Any misclassification of a test as either using an API securely when it is not or the other way around reveals false negatives or positives of the analysis, respectively.

We focus the following discussion on generating test cases from CrySL rules on CrySL-related features (e.g., if an analysis catches all deviations from an object's defined usage pattern). However, one could easily modify CogniCrypt$_{\text{test}}$ to also account for more technical features of analyses (e.g., treatment of method calls or fields, precision, scalability). In this fashion, CogniCrypt$_{\text{test}}$ may be used as a benchmark generator for individual APIs. CogniCrypt$_{\text{test}}$ would likely not benefit Eclipse users a lot.

There are multiple test-case generation strategies CogniCrypt$_{\text{test}}$ could take, depending on the aspired use case. We focus our discussion on generating all possible test cases from a given CrySL rule. We do so because we do not aim to maximize any specific metric such as test coverage or efficiency, but instead aim to show that CogniCrypt$_{\text{test}}$ can indeed generate a comprehensive test suite for an API based on that API's CrySL rules. Other strategies include generating test cases randomly, aiming to maximize some metrics of coverage, following a genetic algorithm, or using learning approaches.

For the purpose of this discussion, we distinguish between aiming at generating test cases *with* and *without* misuses. To approach the discussion more systematically, we also discuss how CogniCrypt$_{\text{test}}$ would treat the different sections of a CrySL rule separately from each other. Figure 9.2 summarizes CogniCrypt$_{\text{test}}$'s workflow. The tool would take as input a set of CrySL rules for an API and first parse them ①. This step leaves CogniCrypt$_{\text{test}}$ with forbidden methods, constraints, state machine, and predicates for any object for whose type CogniCrypt$_{\text{test}}$ has a CrySL rule for.

Forbidden methods are straightforward to address with CogniCrypt$_{\text{test}}$ ②. For a correct test case, CogniCrypt$_{\text{test}}$ would ignore forbidden methods. For a test case that should contain a misuse, CogniCrypt$_{\text{test}}$ would generate a call to such a method.

To fill parameter values ③, CogniCrypt$_{\text{test}}$ could fetch values from the `CONSTRAINTS` section of the CrySL rule under test. If the test case should implement an incorrect use because of an incorrect parameter value, CogniCrypt$_{\text{test}}$ may generate random data of the appropriate data type.

To generate method calls ④, CogniCrypt$_{\text{test}}$ would navigate through the state machine. To generate correct test cases, CogniCrypt$_{\text{test}}$ would transform paths through the state machine that lead to accepting states into code. Any path, on the other hand, that ends in the error state may be transformed into a test cases with such a misuse.

If two classes CogniCrypt$_{\text{fix}}$ has CrySL rules for should be composed, i.e., the test case must generate an object with a valid predicate that is consumed by another object in the test case ⑤, CogniCrypt$_{\text{test}}$ may generate code following the same approach as CogniCrypt$_{\text{gen}}$ (Section 7.1). For test cases with invalid predicates, CogniCrypt$_{\text{test}}$ may follow one of three strategies. First, if an object requires several calls to be used properly, CogniCrypt$_{\text{test}}$

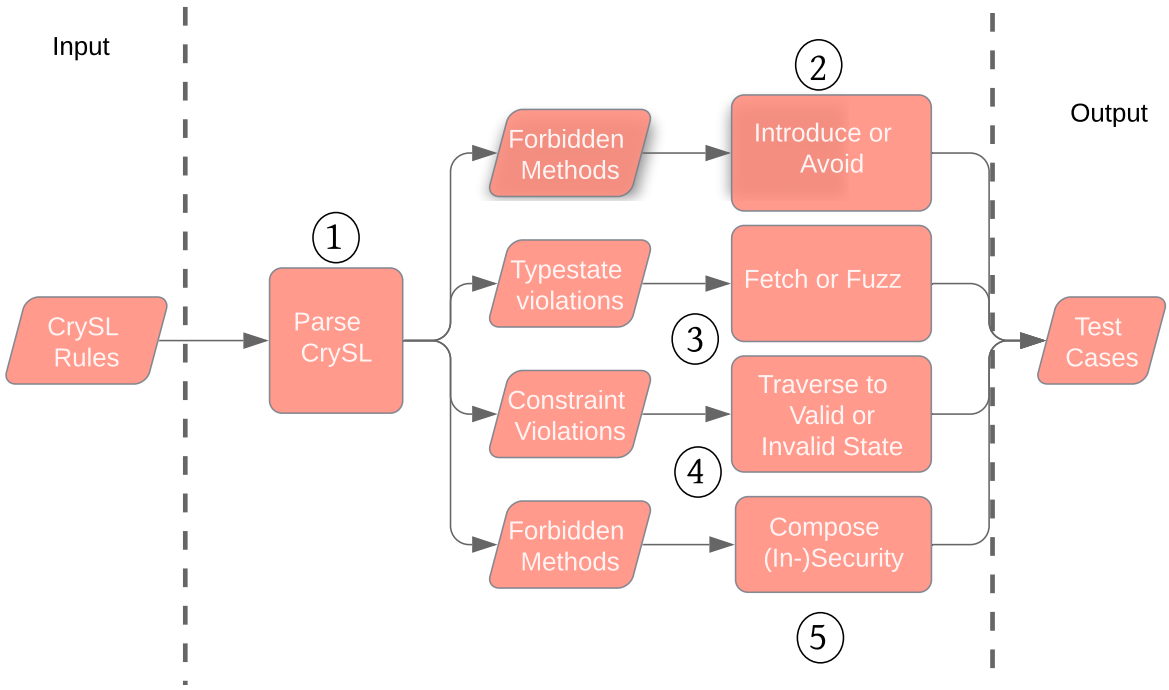

Figure 9.2: The General Workflow of CogniCrypt$_{\text{TEST}}$.

would select the first call in the corresponding usage pattern (e.g., `getInstance(String)` for `KeyGenerator`). It would, however, stop there and not continue generating the remaining calls on that object. As a result, that object does not receive the predicate. Second, if an object does only require one call, but calling another method does kill the predicate (e.g. `PBEKeySpec` in Figure 5.2), CogniCrypt$_{\text{TEST}}$ generates that call, too. Third, should this step fail as well, but the first call on the object, which should ensure the predicate, contain any parameters with constraints on them, CogniCrypt$_{\text{TEST}}$ would generate an object into the test case that does not fulfil its constraint. This way, the object, which should ensure the predicate, does not get one. By supporting the three aforementioned strategies, CogniCrypt$_{\text{TEST}}$ covers all classes in our JCA rule set. We leave it to future work that actually implements CogniCrypt$_{\text{TEST}}$ to explore other rule sets and determine whether or not these three strategies suffice or more are needed.

By following the outlined approach, CogniCrypt$_{\text{TEST}}$ can generate example uses for all CrySL rules it is given. To transform them into test cases, CogniCrypt$_{\text{FIX}}$ would annotate them with expected test results. That is, it marks any test case that, for instance, contains a forbidden call as incorrect for containing that forbidden call. As CogniCrypt$_{\text{TEST}}$ generates the example uses to begin with and is hence aware of which example uses are incorrect for which reasons, it can on-the-fly generate the annotations along with them.

9.4 CogniCrypt$_{\text{doc}}$ – Generating documentation for hard-to-use APIs

Our final suggestion is CogniCrypt$_{\text{DOC}}$. CogniCrypt$_{\text{DOC}}$ addresses the issue of insufficient documentation raised and discussed in a lot of work on the usability of Crypto APIs. Many developers are overwhelmed by both the complexity of and the domain knowledge required to understand Crypto APIs [ABF+17, MKW18, PHR19, NKMB16, HZH19]. While CrySL rules themselves may serve as documentation, most people likely prefer running text in the form of manuals akin to the JCA Reference Guide [Inc17] or method documentation (e.g., JavaDoc)

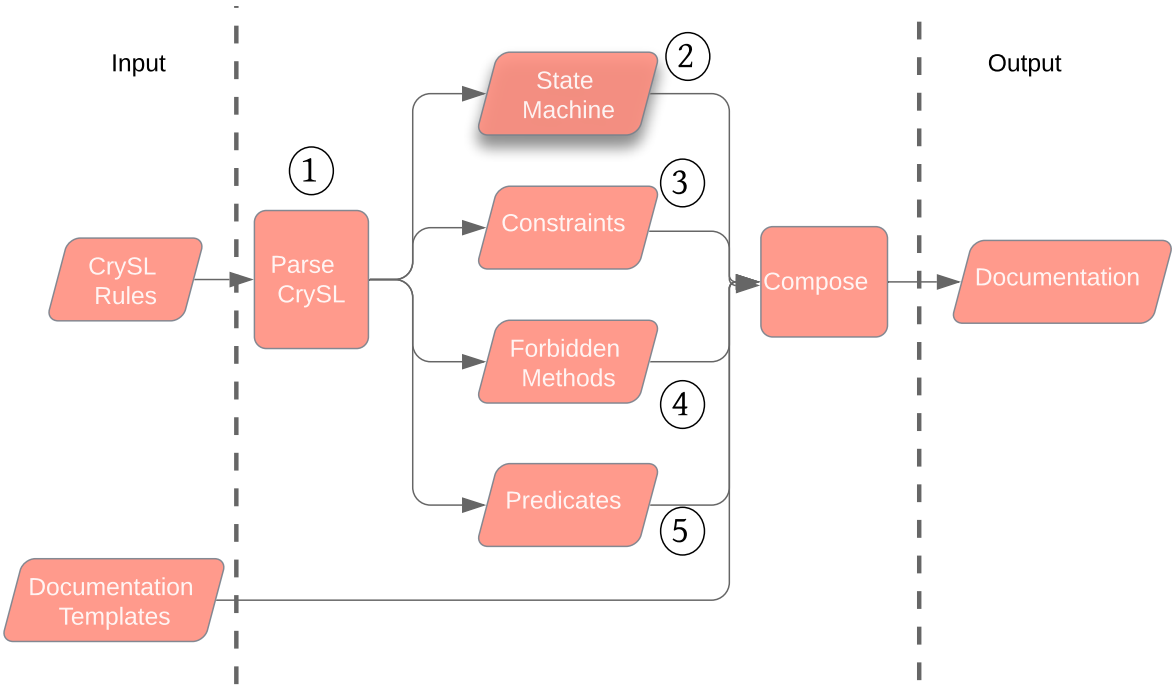

Figure 9.3: The General Workflow of CogniCrypt$_{DOC}$.

over specifications. As Huesmann et al. [HZH19] point out, having multiple media that target different groups is beneficial in any case. CogniCrypt$_{DOC}$ may generate such documentation from a CrySL rule.

In the following, we illustrate CogniCrypt$_{DOC}$'s workflow in Figure 9.3 by discussing how CogniCrypt$_{DOC}$ would generate full-fledged manuals instead of other kinds of documentation such as method documentation. To this end, we return to `PBEKeySpec` from the JCA, we have used throughout this thesis as an example. The official documentation of each class in the JCA contains a few paragraphs of introduction, followed by method documentation of all methods in the class. The introduction for `PBEKeySpec` [Ora19b], which we show in Figure 9.4, briefly describes what the class' purpose is and refers to a few adjacent cryptographic standards. It also lays out the reasoning for not storing the password object as a `String`, but instead a `char` array. The introduction does not, however, link any of this explanation to the class' methods. It also does not discuss any of the other parameters of `PBEKeySpec`'s constructors beyond the password. We aim with CogniCrypt$_{DOC}$ to generate an extension to the introduction akin to what is shown in Figure 9.5 that accomplishes to discuss these two things.

CogniCrypt$_{DOC}$ would receive a set of CrySL rules, including the one for `PBEKeySpec` (Figure 5.2) as well as *documentation text templates* as input that provide the boilerplate text CogniCrypt$_{DOC}$ uses to generate the full documentation. CogniCrypt$_{DOC}$ would come with one template for each language construct in CrySL so that the templates are not class-specific. Templates contain placeholders, though that must be filled with information from the CrySL rules. All italic words in the text in Figure 9.5 represent former placeholders CogniCrypt$_{DOC}$ has replaced with information from `PBEKeySpec`'s CrySL rule during the generation process. CogniCrypt$_{DOC}$ would traverse a given CrySL rule from top to bottom. To generate Line 290, CogniCrypt$_{DOC}$ would take the boilerplate text from Figure 9.6 and replace the two placeholders $number and $class_name with two—the number of defined methods in the rule's **`EVENTS`** section—and `PBEKeySpec`—the class' name—respectively.

To document the two required methods of `PBEKeySpec` (Lines 291–292), CogniCrypt$_{DOC}$

```
A user-chosen password that can be used with password-based encryption (PBE).

The password can be viewed as some kind of raw key material, from which the
encryption mechanism that uses it derives a cryptographic key.

Different PBE mechanisms may consume different bits of each password character.
For example, the PBE mechanism defined in PKCS #5 looks at only the low order 8
bits of each character, whereas PKCS #12 looks at all 16 bits of each character.

You convert the password characters to a PBE key by creating an instance of the
appropriate secret-key factory. For example, a secret-key factory for PKCS #5
will construct a PBE key from only the low order 8 bits of each password
character, whereas a secret-key factory for PKCS #12 will take all 16 bits of
each character.

Also note that this class stores passwords as char arrays instead of String
objects (which would seem more logical), because the String class is immutable
and there is no way to overwrite its internal value when the password stored in
it is no longer needed. Hence, this class requests the password as a char
array, so it can be overwritten when done.
```

Figure 9.4: Introduction to Official PBEKeySpec Documentation in JDK 13[Ora19b]

```
290 There are two methods in PBEKeySpec that may be called to use the class
    correctly.
291 First, method PBEKeySpec(char[], byte[], int, int) must be called exactly once.
292 Last, method clearPassword() must be called exactly once.
293
294 Three parameters of method PBEKeySpec(char[] password, byte[]  salt, int, int) are
    constrained in one way or another.
295 Any data that is part of char[] password must never be a String at any point
    in the program.
296 The parameter Byte[] salt must be randomized.
297 The parameter int iterationCount must be at least 10,000.
298
299 Do not use method PBEKeySpec(char[]). Instead use method PBEKeySpec(char[], byte[],
    int, int).
300
301 The class ensures that its own instance can be used as a specced Key after
    PBEKeySpec(char[], byte[], int, int). This is no longer ensured after method
    clearPassword() is called.
```

Figure 9.5: Result for PBEKeySpec

```
There are $number methods in $class_name that may be called to use the class correctly.
```

Figure 9.6: Text Template for Required Method Calls

```
$Position, method $method_name $must_may be called $restriction $cardinality.
```

Figure 9.7: Text Template for Required Method Calls

would navigate through `PBEKeySpec`'s state machine and apply the template in Figure 9.7 to each transition. In case the state machine branches, CogniCrypt$_{DOC}$ would make use of specific templates that account for all available branches.

CogniCrypt$_{DOC}$ may apply similar templates to all constraints of the rule and predicates parameter objects in the rule require, leading to Lines 294–297. Take the parameter `iterationCount` that must be, according to Line 83 of `PBEKeySpec`'s CrySL rule in Figure 5.2, greater than or equal to 10,000. CogniCrypt$_{DOC}$ would insert, for the purpose of generating the documentation, the variable name `iterationCount` and the restriction on it into the corresponding text template. This insertion leads to the explanation in Line 296. CogniCrypt$_{DOC}$ would support for each type of constraint with specific templates (e.g., required predicate in Line 296 or built-in predicate `neverTypeOf` in Line 295).

`PBEKeySpec`'s CrySL rule also defines a constructor as *forbidden* (Line 87 in Figure 5.2). For the documentation, CogniCrypt$_{DOC}$ would generate Line 299 to warn of the constructor's use. As the CrySL rule suggests an alternative, CogniCrypt$_{DOC}$ generates the second sentence in the line recommending to use this alternative instead.

Finally, for predicates the class *ensures* or *negates*, CogniCrypt$_{DOC}$ would generate Line 301 in the example. To this end, it applies the variable the predicate is ensured upon, in the case of `PBEKeySpec` that is `this`, the name of the predicate `speccedKey` and the method after which this predicate is ensured to a predicate text template. In the case of `speccedKey`, the `after` keyword is used to define the method after whose call the predicate should hold. CogniCrypt$_{DOC}$ would hence insert the name of this method into the documentation. For rules, where the keyword `after` is not used, CogniCrypt$_{DOC}$ may instead use the names of methods that lead to an accepting state in the finite state machine.

Following this approach, CogniCrypt$_{DOC}$ is capable of providing documentation that does actually explain how to use a class. The design we present here can easily be extended to also address composition with other classes or to include correct usage examples generated with CogniCrypt$_{GEN}$. We also believe that CogniCrypt users would benefit from an integration of CogniCrypt$_{DOC}$ into the plugin such that it display CogniCrypt$_{DOC}$-provided documentation.

9.5 Conclusion

In this chapter, we have discussed four applications of CrySL that go beyond CogniCrypt$_{SAST}$ and CogniCrypt$_{GEN}$. The first one – CryptoOracle – provides developers direct support in their use of cryptographic APIs, independent of any tools. The other three are rather design sketches of yet unrealized types of tool support, built on top of CrySL. We have presented only rough sketches of each of the three approaches and did not cover each and every edge case. This is because we wanted to illustrate the ideas underlying each of the three approaches, not provide detailed plans for their implementations. All four application ideas demonstrate the expressive power of CrySL. Although originally designed as a basis of CogniCrypt$_{SAST}$, CrySL's expressiveness suffices to serve as a building block for API tool support that goes beyond the well-trodden path of program analysis.

10 Conclusion

Cryptography can help secure sensitive data, but only if applications use cryptographic components securely. The cryptography domain comprises a wide range of algorithms, each can be configured in multiple ways. On top of that, developers need to keep in mind even more complex requirements such as that certain keys may not be proper for certain operations or how keys are properly and securely derived from a password. All this becomes an issue for developers with little or no experience in cryptography and prior research shows that this setup does not work.

In this thesis, we have presented CogniCrypt, an integrated approach to addressing this issue. With CogniCrypt, we support application developers in using cryptographic APIs by relieving them of having to know how to use an API. Our first contribution, the API-usage specification language CrySL, lies at the core of this effort. The target audience of CrySL are cryptography experts. Using CrySL, they can specify how their APIs should be used. To show CrySL's practicality, we have modelled the most popular Crypto API, the Java Cryptography Architecture (JCA), in CrySL. Over the course of our work, other APIs have been modelled in CrySL as well.

CogniCrypt$_{\text{SAST}}$, the second contribution of this thesis, transforms CrySL rules into a static program analysis. Bootstrapped with the JCA rule set, we have conducted three empirical evaluations. We found CogniCrypt$_{\text{SAST}}$ to be very precise. We were also able to confirm previous studies' results on cryptographic misuse in Android and were the first to analyze the entirety of Maven Central for Crypto misuses within just a handful of days.

Third, we contribute another type of CrySL-based tool support as part of this thesis: the code generator for cryptographic use cases CogniCrypt$_{\text{GEN}}$. On top of CrySL rules, CogniCrypt$_{\text{GEN}}$ consumes Java code templates. These templates implement non-security-critical wrapper code for a cryptographic use case and further encode what cryptographic code should be generated how from CrySL rules. Our empirical evaluation revealed CogniCrypt$_{\text{GEN}}$ to be very usable and its artefacts compact and maintainable.

In our fourth contribution, we empirically evaluated CogniCrypt as a whole. In particular, our experiment examined whether or not CogniCrypt keeps its promise to effectively help developers in avoiding cryptographic misuse. We have implemented CogniCrypt as an Eclipse plugin consisting of both CogniCrypt$_{\text{SAST}}$ and CogniCrypt$_{\text{GEN}}$, integrated with the Eclipse plugin framework. We asked 18 participants to implement small cryptographic tasks with and without CogniCrypt. Our results reveal that CogniCrypt facilitates faster, more functional, and more secure development of cryptographic code. The experiment provides evidence that CogniCrypt does indeed combat cryptographic misuse effectively.

Finally, we have closed out the thesis by outlining four other approaches that build on top

of CrySL, three of which have yet to be implemented. All four approaches help application developers further when interacting with Crypto APIs. However, there are more directions than these four approaches CogniCrypt ought to be taken. One such direction targets CrySL's relation to other specification languages that cover subsets of CrySL. When two languages overlap in the properties they can express, specifying the same issue in both languages to receive their accompanying tool support is redundant. To save specification overhead when developing rules in each of those languages, one could develop adapters between them that transform specifications in one language into specifications of the other. This way, one could specify something in one language and have it automatically transformed into the other. The most straightforward concern, however, is coverage in several dimensions. First, there are still other Crypto APIs that have not yet been modelled with CrySL. Second, there are also non-crypto APIs that are hard to use and that one might attempt to model in CrySL as well. Just because CrySL was originally targeted at Crypto APIs does not mean, the language may not be applied to other APIs as well. Third, CogniCrypt and CrySL may be ported to other popular programming languages (e.g., C++ or Python). An approach as effective as CogniCrypt in addressing API misuse should not only be maintained for the settings it has already been developed for, but should actively be expanded. That would allow other platforms to reap the same benefits CogniCrypt brings to Java Crypto APIs already.

Bibliography

[AAC+05] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie J. Hendren, Sascha Kuzins, Ondrej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding trace matching with free variables to aspectj. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16-20, 2005, San Diego, CA, USA*, pages 345–364, 2005.

[ABB+09] Dima Alhadidi, Amine Boukhtouta, Nadia Belblidia, Mourad Debbabi, and Prabir Bhattacharya. The dataflow pointcut: a formal and practical framework. In *Proceedings of the 8th International Conference on Aspect-Oriented Software Development, AOSD 2009, Charlottesville, Virginia, USA, March 2-6, 2009*, pages 15–26, 2009.

[ABF+16] Yasemin Acar, Michael Backes, Sascha Fahl, Doowon Kim, Michelle L. Mazurek, and Christian Stransky. You get where you're looking for: The impact of information sources on code security. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*, pages 289–305, 2016.

[ABF+17] Yasemin Acar, Michael Backes, Sascha Fahl, Simson L. Garfinkel, Doowon Kim, Michelle L. Mazurek, and Christian Stransky. Comparing the usability of cryptographic apis. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, pages 154–171, 2017.

[ABKT16] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. Androzoo: collecting millions of android apps for the research community. In *Proceedings of the 13th International Conference on Mining Software Repositories, MSR 2016, Austin, TX, USA, May 14-22, 2016*, pages 468–471, 2016.

[ABP+13] Nadhem AlFardan, Daniel J. Bernstein, Kenneth G. Paterson, Bertram Poettering, and Jacob C. N. Schuldt. On the security of RC4 in TLS. In *USENIX Security Symposium*, pages 305–320, 2013.

[AC18] Hala Assal and Sonia Chiasson. Security in the software development lifecycle. In *Fourteenth Symposium on Usable Privacy and Security, SOUPS 2018, Baltimore, MD, USA, August 12-14, 2018.*, pages 281–296, 2018.

[ANA+15] Steven Arzt, Sarah Nadi, Karim Ali, Eric Bodden, Sebastian Erdweg, and Mira Mezini. Towards secure integration of cryptographic software. In *SIGPLAN Symposium on New Ideas in Programming and Reflections on Software at SPLASH (Onward!)*, 2015.

[ARF+14] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick D. McDaniel. Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, pages 259–269, 2014.

[Art18] Philippe Arteau. Findsecbugs, 2018.

[ASLRD99] Alyce S. Adams, Stephen B. Soumerai, Jonathan Lomas, and Dennis Ross-Degnan. Evidence of self-report bias in assessing adherence to guidelines. *International Journal for Quality in Health Care*, 11(3):187–192, 06 1999.

[ASW+17] Yasemin Acar, Christian Stransky, Dominik Wermke, Charles Weir, Michelle L. Mazurek, and Sascha Fahl. Developers need support, too: A survey of security advice for software developers. In *SecDev*, pages 22–26. IEEE Computer Society, 2017.

[BA07] Kevin Bierhoff and Jonathan Aldrich. Modular typestate checking of aliased objects. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*, pages 301–320, 2007.

[BBG+63] John W. Backus, Friedrich L. Bauer, Julien Green, C. Katz, John McCarthy, Alan J. Perlis, Heinz Rutishauser, Klaus Samelson, Bernard Vauquois, Joseph Henry Wegstein, Adriaan van Wijngaarden, Michael Woodger, and Peter Naur. Revised report on the algorithm language ALGOL 60. *Communications of the ACM*, 6(1):1–17, 1963.

[BC14] Rebecca Balebako and Lorrie Faith Cranor. Improving app privacy: Nudging app developers to protect user privacy. *IEEE Security & Privacy*, 12(4):55–58, 2014.

[BD16] Alexandre Melo Braga and Ricardo Dahab. Mining cryptography misuse in online forums. In *2016 IEEE International Conference on Software Quality, Reliability and Security, QRS 2016, Companion, Vienna, Austria, August 1-3, 2016*, pages 143–150, 2016.

[BDA+17] Alexandre Melo Braga, Ricardo Dahab, Nuno Antunes, Nuno Laranjeiro, and Marco Vieira. Practical evaluation of static analysis tools for cryptography: Benchmarking method and case study. In *28th IEEE International Symposium on Software Reliability Engineering, ISSRE 2017, Toulouse, France, October 23-26, 2017*, pages 170–181, 2017.

[BKS+18] Eric Bodden, Stefan Krueger, Johannes Spaeth, Karim Ali, and Mira Mezini. CVE-2018-12240. Available from Symantec, CVE-ID CVE-2018-12240., August 3 2018.

[BLH12] Eric Bodden, Patrick Lam, and Laurie Hendren. Partially evaluating finite-state runtime monitors ahead of time. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 34(2):7:1–7:52, June 2012.

[BLS12] Daniel J. Bernstein, Tanja Lange, and Peter Schwabe. The security impact of a new cryptographic library. In *Progress in Cryptology - LATINCRYPT 2012 - 2nd International Conference on Cryptology and Information Security in Latin America, Santiago, Chile, October 7-10, 2012. Proceedings*, pages 159–176, 2012.

[Bod10] Eric Bodden. Efficient hybrid typestate analysis by determining continuation-equivalent states. In *ICSE '10: International Conference on Software Engineering*, pages 5–14, New York, NY, USA, May 2010. ACM.

[Bod14] Eric Bodden. TS4J: a fluent interface for defining and computing typestate analyses. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State Of the Art in Java Program analysis, SOAP 2014, Edinburgh, UK, Co-located with PLDI 2014, June 12, 2014*, pages 1:1–1:6, 2014.

[Bod18] Eric Bodden. The secret sauce in efficient and precise static analysis: the beauty of distributive, summary-based static analyses (and how to master them). In *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops, ISSTA 2018, Amsterdam, Netherlands, July 16-21, 2018*, pages 85–93, 2018.

[BPF19] Maximilian Blochberger, Tom Petersen, and Hannes Federrath. Mitigating cryptographic mistakes by design. In *Mensch und Computer 2019 - Workshopband, Hamburg, Germany, September 8-11, 2019*, 2019.

[BR01] Thomas Ball and Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. In *Model Checking Software, 8th International SPIN Workshop, Toronto, Canada, May 19-20, 2001, Proceedings*, pages 103–122, 2001.

[BR02] Thomas Ball and Sriram K. Rajamani. The SLAM project: debugging system software via static analysis. In *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, January 16-18, 2002*, pages 1–3, 2002.

[Bro96] John Brooke. SUS-A quick and dirty usability scale. *Usability Evaluation in Industry*, 189(194):4–7, 1996.

[Buc16] Johannes A. Buchmann. *Einführung in die Kryptographie, 6. Auflage*. Springer, 2016.

[BW12] Raymond P. L. Buse and Westley Weimer. Synthesizing API usage examples. In *International Conference on Software Engineering (ICSE)*, pages 782–792, 2012.

[CGK12] Gary Charness, Uri Gneezy, and Michael A Kuhn. Experimental methods: Between-subject and within-subject design. *Journal of Economic Behavior & Organization*, 81(1):1–8, 2012.

[CNKX16] Alexia Chatzikonstantinou, Christoforos Ntantogian, Georgios Karopoulos, and Christos Xenakis. Evaluation of cryptography usage in android applications. In *International Conference on Bio-inspired Information and Communications Technologies*, pages 83–90, 2016.

[Cos] Cossack Labs. Themis. `https://www.cossacklabs.com/themis/`.

[DGC95] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *ECOOP'95 - Object-Oriented*

Programming, 9th European Conference, Århus, Denmark, August 7-11, 1995, Proceedings, pages 77–101, 1995.

[DH79] Whitfield Diffie and Martin E Hellman. Privacy and authentication: An introduction to cryptography. *Proceedings of the IEEE*, 67(3):397–427, 1979.

[DR98] Joan Daemen and Vincent Rijmen. The block cipher rijndael. In *CARDIS*, volume 1820 of *Lecture Notes in Computer Science*, pages 277–284. Springer, 1998.

[DR99] Joan Daemen and Vincent Rijmen. Aes proposal: Rijndael. 1999.

[DSZ15] Daniel Demmler, Thomas Schneider, and Michael Zohner. ABY - A framework for efficient mixed-protocol secure two-party computation. In *NDSS*. The Internet Society, 2015.

[EBFK13] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. An empirical study of cryptographic misuse in android applications. In *ACM Conference on Computer and Communications Security*, pages 73–84, 2013.

[EMST78] William F Ehrsam, Carl HW Meyer, John L Smith, and Walter L Tuchman. Message verification and transmission error detection by block chaining, February 14 1978. US Patent 4,074,066.

[FBX+17] Felix Fischer, Konstantin Böttinger, Huang Xiao, Christian Stransky, Yasemin Acar, Michael Backes, and Sascha Fahl. Stack overflow considered harmful? the impact of copy&paste on android application security. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, pages 121–136, 2017.

[Fei19] Johannes Feichtner. A comparative study of misapplied crypto in android and ios applications. In *Proceedings of the 16th International Joint Conference on e-Business and Telecommunications, ICETE 2019 - Volume 2: SECRYPT, Prague, Czech Republic, July 26-28, 2019*, pages 96–108, 2019.

[FHM+12] Sascha Fahl, Marian Harbach, Thomas Muders, Matthew Smith, Lars Baumgärtner, and Bernd Freisleben. Why Eve and Mallory love Android: an Analysis of Android SSL (In)security. In *ACM Conference on Computer and Communications Security*, pages 50–61, 2012.

[fISB17] German Federal Office for Information Security (BSI). Cryptographic mechanisms: Recommendations and key lengths. Technical Report BSI TR-02102-1, BSI, January 2017.

[fISB19] German Federal Office for Information Security (BSI). Cryptographic mechanisms: Recommendations and key lengths. Technical Report BSI TR-02102-2, BSI, February 2019.

[FLW12] Christian Forler, Stefan Lucks, and Jakob Wenzel. Designing the API for a cryptographic library - A misuse-resistant application programming interface. In *Reliable Software Technologies - Ada-Europe 2012 - 17th Ada-Europe International Conference on Reliable Software Technologies, Stockholm, Sweden, June 11-15, 2012. Proceedings*, pages 75–88, 2012.

[Fow05] Martin Fowler. FluentInterface, 2005. `https://martinfowler.com/bliki/FluentInterface.html`.

[FXK+19] Felix Fischer, Huang Xiao, Ching-yu Kao, Yannick Stachelscheid, Benjamin Johnson, Danial Razar, Paul Fawkesley, Nat Buckley, Konstantin Böttinger, Paul Muntean, and Jens Grossklags. Stack overflow considered helpful! deep learning security nudges towards stronger cryptography. In *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, pages 339–356, 2019.

[Gam84] Taher El Gamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In *CRYPTO*, volume 196 of *Lecture Notes in Computer Science*, pages 10–18. Springer, 1984.

[Gao05] Lei Gao. Latin squares in experimental design. 2005.

[GGF17] Paul Grassi, Michael Garcia, and James Fenton. Digital identity guidelines. Technical report, National Institute of Standards and Technology, 2017.

[GIJ+12] Martin Georgiev, Subodh Iyengar, Suman Jana, Rishita Anubhai, Dan Boneh, and Vitaly Shmatikov. The most dangerous code in the world: Validating SSL certificates in non-browser software. In *Conference on Computer and Communications Security (CCS)*, pages 38–49, 2012.

[GIW+] Peter Leo Gorski, Luigi Lo Iacono, Dominik Wermke, Christian Stransky, Sebastian Möller, Yasemin Acar, and Sascha Fahl. Developers deserve security warnings, too: On the effect of integrated security advice on cryptographic API misuse. In *Fourteenth Symposium on Usable Privacy and Security, SOUPS 2018, Baltimore, MD, USA, August 12-14, 2018*, pages 265–281.

[GKL+19] Jun Gao, Pingfan Kong, Li Li, Tegawendé F. Bissyandé, and Jacques Klein. Negative results on mining crypto-api usage rules in android apps. In *Proceedings of the 16th International Conference on Mining Software Repositories, MSR 2019, 26-27 May 2019, Montreal, Canada*, pages 388–398, 2019.

[GOA05] Simon Goldsmith, Robert O'Callahan, and Alexander Aiken. Relational queries over program traces. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16-20, 2005, San Diego, CA, USA*, pages 385–402, 2005.

[Goo] Google LLC. Google tink. `https://github.com/google/tink`.

[GS16] Matthew Green and Matthew Smith. Developers are not the enemy!: The need for usable security apis. *IEEE Security & Privacy*, 14(5):40–46, 2016.

[GTM14] Ricardo Rodriguez Garcia, Julie Thorpe, and Miguel Vargas Martin. Cryptoassistant: Towards facilitating developer's encryption of sensitive data. In *2014 Twelfth Annual International Conference on Privacy, Security and Trust, Toronto, ON, Canada, July 23-24, 2014*, pages 342–346, 2014.

[GWL+19] Zuxing Gu, Jiecheng Wu, Jiaxiang Liu, Min Zhou, and Ming Gu. An empirical study on api-misuse bugs in open-source C programs. In *43rd IEEE Annual Computer Software and Applications Conference, COMPSAC 2019, Milwaukee, WI, USA, July 15-19, 2019, Volume 1*, pages 11–20, 2019.

[HCP+95] James R Hebert, Lynn Clemow, Lori Pbert, Ira S Ockene, and Judith K Ockene. Social desirability bias in dietary self-report may compromise the validity of dietary intake measures. *International journal of epidemiology*, 24(2):389–398, 1995.

[Hec14] Philipp C. Heckel. CipherInputStream for AEAD modes is insecure in JDK7 (GCM, EAX, etc.), 2014.

[HG14] Ekawat Homsirikamol and Kris Gaj. Can high-level synthesis compete against a hand-written code in the cryptographic domain? A case study. In *2014 International Conference on ReConFigurable Computing and FPGAs, ReConFig14, Cancun, Mexico, December 8-10, 2014*, pages 1–8, 2014.

[HGT17] Julie M. Haney, Simson L. Garfinkel, and Mary Frances Theofanos. Organizational practices in cryptographic development and testing. In *2017 IEEE Conference on Communications and Network Security, CNS 2017, Las Vegas, NV, USA, October 9-11, 2017*, pages 1–9, 2017.

[Hol19] Malte Hollmann. From CrySL to an Advanced Crypto Library. Master's thesis, Paderborn University, 2019.

[Hou09] Russell Housley. Cryptographic message syntax (CMS). *RFC*, 5652:1–56, 2009.

[HTAP18] Julie M. Haney, Mary Theofanos, Yasemin Acar, and Sandra Spickard Prettyman. "we make it a big deal in the company": Security mindsets in organizations that develop cryptographic products. In *Fourteenth Symposium on Usable Privacy and Security, SOUPS 2018, Baltimore, MD, USA, August 12-14, 2018*, pages 357–373, 2018.

[HZH19] Rolf Huesmann, Alexander Zeier, and Andreas Heinemann. Eigenschaften optimierter api-dokumentationen im entwicklungsprozess sicherer software. In *Mensch und Computer 2019 - Workshopband, Hamburg, Germany, September 8-11, 2019*, 2019. In German.

[IG17] Luigi Lo Iacono and Peter Leo Gorski. I do and i understand. not yet true for security apis. so sad. In *Proceedings 2nd European Workshop on Usable Security. Internet Society. https://doi. org/10.14722/eurousec*, 2017.

[IKND16] Soumya Indela, Mukul Kulkarni, Kartik Nayak, and Tudor Dumitras. Toward semantic cryptography apis. In *IEEE Cybersecurity Development, SecDev 2016, Boston, MA, USA, November 3-4, 2016*, pages 9–14, 2016.

[Inc17] Oracle Inc. Java cryptography architecture (JCA), 2017. `https://docs.oracle.com/javase/8/docs/technotes/guides/security/crypto/CryptoSpec.html`.

[JKC18] Swati Jaiswal, Uday P. Khedker, and Supratik Chakraborty. Demand-driven alias analysis : Formalizing bidirectional analyses for soundness and precision. *CoRR*, abs/1802.00932, 2018.

[JSM+19] Paulius Juodisius, Atrisha Sarkar, Raghava Rao Mukkamala, Michal Antkiewicz, Krzysztof Czarnecki, and Andrzej Wasowski. Clafer: Lightweight modeling of structure, behaviour, and variability. *Programming Journal*, 3(1):2, 2019.

[JSMB13] Brittany Johnson, Yoonki Song, Emerson R. Murphy-Hill, and Robert W. Bowdidge. Why don't software developers use static analysis tools to find bugs? In *ICSE*, pages 672–681. IEEE Computer Society, 2013.

[KAB20] Stefan Krüger, Karim Ali, and Eric Bodden. CogniCrypt$_{GEN}$ - Generating Code for the Secure Usage of Crypto APIs. In *Internationl Symposium on Code Generation and Optimization (CGO)*, 2020.

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William Griswold. An overview of aspectj. *ECOOP 2001—Object-Oriented Programming*, pages 327–354, 2001.

[KLCL18] Christopher Kane, Bo Lin, Saksham Chand, and Yanhong A. Liu. High-level cryptographic abstractions. *CoRR*, abs/1810.09065, 2018.

[KLHK10] Jinhan Kim, Sanghoon Lee, Seung-won Hwang, and Sunghun Kim. Towards an intelligent code search engine. 2010.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer, 1997.

[KNR+17] Stefan Krüger, Sarah Nadi, Michael Reif, Karim Ali, Mira Mezini, Eric Bodden, Florian Göpfert, Felix Günther, Christian Weinert, Daniel Demmler, and Ram Kamath. CogniCrypt: Supporting Developers in Using Cryptography. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, pages 931–936, 2017.

[KNR+21] Stefan Krüger, Sarah Nadi, Michael Reif, Anna-Katharina Wickert, Rodrigo Bonifácio, Karim Ali, Sascha Fahl, Yasemine Acar, Eric Bodden, and Mira Mezini. The Effects of Use-Case Based Code Generation on Security Code. In *International Conference on Software Engineering (ICSE)*, 2021.

[KRZ14] Iman Keivanloo, Juergen Rilling, and Ying Zou. Spotting working code examples. In *International Conference on Software Engineering (ICSE)*, pages 664–675, 2014.

[KSA+18] Stefan Krüger, Johannes Späth, Karim Ali, Eric Bodden, and Mira Mezini. CrySL: An Extensible Approach to Validating the Correct Usage of Cryptographic APIs. In *European Conference on Object-Oriented Programming (ECOOP)*, 2018.

[KSA+19a] Stefan Krueger, Johannes Spaeth, Karim Ali, Eric Bodden, and Mira Mezini. CrySL Rule Set for JCA, 2019. `https://github.com/CROSSINGTUD/Crypto-API-Rules`.

[KSA+19b] Stefan Krüger, Johannes Späth, Karim Ali, Eric Bodden, and Mira Mezini. CrySL: An Extensible Approach to Validating the Correct Usage of Cryptographic APIs. In *IEEE Transactions on Software Engineering (TSE)*, 2019.

[Lab19] Information Technology Laboratory. FIPS 140-2, annex a: Approved security functions for fips pub 140-2, security requirements for cryptographic modules, 2019.

[LBLH11] Patrick Lam, Eric Bodden, Ondřej Lhoták, and Laurie Hendren. The Soot framework for Java program analysis: a retrospective. In *Cetus Users and Compiler Infrastructure Workshop (CETUS 2011)*, October 2011.

[LCWZ14] David Lazar, Haogang Chen, Xi Wang, and Nickolai Zeldovich. Why does cryptographic software fail?: a case study and open problems. In *ACM Asia-Pacific Workshop on Systems (APSys)*, pages 7:1–7:7, 2014.

[Leg18] Legion of the Bouncy Castle Inc. BouncyCastle, 2018. `https://www.bouncycastle.org/java.html`.

[Leu19] Sebastian Leuer. Sharper Crypto-API Analysis: Entwicklung eines integrierten Plugins zur statischen Code-Analyse der Nutzung von Krypto-APIs in C#. Master's thesis, Hochschule Düsseldorf, 2019.

[LL05] V. Benjamin Livshits and Monica S. Lam. Finding security vulnerabilities in java applications with static analysis. In *Proceedings of the 14th USENIX Security Symposium, Baltimore, MD, USA, July 31 - August 5, 2005*, 2005.

[Lou] Loup Vaillant. Monocypher. `https://monocypher.org/`.

[LST+19] Tamara Lopez, Helen Sharp, Thein Than Tun, Arosha K. Bandara, Mark Levine, and Bashar Nuseibeh. "hopefully we are mostly secure": views on secure code in professional practice. In *Proceedings of the 12th International Workshop on Cooperative and Human Aspects of Software Engineering, CHASE@ICSE 2019, Montréal, QC, Canada, 27 May 2019*, pages 61–68, 2019.

[LZLG14] Yong Li, Yuanyuan Zhang, Juanru Li, and Dawu Gu. icryptotracer: Dynamic analysis on misuse of cryptography functions in ios applications. In *Network and System Security - 8th International Conference, NSS 2014, Xi'an, China, October 15-17, 2014, Proceedings*, pages 349–362, 2014.

[Mah16] Khalid Mahmood. Do people overestimate their information literacy skills? a systematic review of empirical evidence on the dunning-kruger effect. *Communications in Information Literacy*, 10(2):3, 2016.

[Mai] MaidSafe. Rust sodium. `https://github.com/maidsafe/rust_sodium`.

[MBB18] Ildar Muslukhov, Yazan Boshmaf, and Konstantin Beznosov. Source attribution of cryptographic API misuse in android applications. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security, AsiaCCS 2018, Incheon, Republic of Korea, June 04-08, 2018*, pages 133–146, 2018.

[MBP+15] Laura Moreno, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Andrian Marcus. How can I use this method? In *International Conference on Software Engineering (ICSE)*, pages 880–890, 2015.

[MK19] Duncan Mitchell and Johannes Kinder. A formal model for checking cryptographic API usage in javascript. In *Computer Security - ESORICS 2019 - 24th European Symposium on Research in Computer Security, Luxembourg, September 23-27, 2019, Proceedings, Part I*, pages 341–360, 2019.

[MKW18] Kai Mindermann, Philipp Keck, and Stefan Wagner. How usable are rust cryptography apis? In *2018 IEEE International Conference on Software Quality,*

Reliability and Security, QRS 2018, Lisbon, Portugal, July 16-20, 2018, pages 143–154, 2018.

[MLL05] Michael C. Martin, V. Benjamin Livshits, and Monica S. Lam. Finding application errors and security flaws using PQL: a program query language. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16-20, 2005, San Diego, CA, USA*, pages 365–383, 2005.

[MLLD16] Siqi Ma, David Lo, Teng Li, and Robert H. Deng. Cdrep: Automatic repair of cryptographic misuses in android applications. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2016, Xi'an, China, May 30 - June 3, 2016*, pages 711–722, 2016.

[MM12] Fabio Martinelli and Ilaria Matteucci. A framework for automatic generation of security controller. *Softw. Test., Verif. Reliab.*, 22(8):563–582, 2012.

[MV04] David McGrew and John Viega. The galois/counter mode of operation (gcm). *submission to NIST Modes of Operation Process*, 20, 2004.

[MVW07] Clint Morgan, Kris De Volder, and Eric Wohlstadter. A static aspect language for checking design rules. In *Proceedings of the 6th International Conference on Aspect-Oriented Software Development, AOSD 2007, Vancouver, British Columbia, Canada, March 12-16, 2007*, pages 63–72, 2007.

[MW18] Kai Mindermann and Stefan Wagner. Usability and security effects of code examples on crypto apis. In *16th Annual Conference on Privacy, Security and Trust, PST 2018, Belfast, Northern Ireland, Uk, August 28-30, 2018*, pages 1–2, 2018.

[Nat18] National Cyber Security Centre. Password administration for system owners. Technical report, National Cyber Security Centre, 2018.

[NCC18] NCCGroup. Visualcodegrepper, 2018.

[NDG+19] Alena Naiakshina, Anastasia Danilova, Eva Gerlitz, Emanuel von Zezschwitz, and Matthew Smith. "if you want, I can store the encrypted password": A password-storage field study with freelance developers. page 140, 2019.

[NDGS20] Alena Naiakshina, Anastasia Danilova, Eva Gerlitz, and Matthew Smith. On Conducting Security Developer Studies with CS Students: Examining a Password-Storage Study with CS Students, Freelancers, and Company Developers. In *CHI*. ACM, 2020. To appear.

[NDT+17] Alena Naiakshina, Anastasia Danilova, Christian Tiefenau, Marco Herzog, Sergej Dechand, and Matthew Smith. Why do developers get password storage wrong?: A qualitative usability study. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 311–328, 2017.

[NDTS18] Alena Naiakshina, Anastasia Danilova, Christian Tiefenau, and Matthew Smith. Deception task design in developer password studies: Exploring a student sample. In *Fourteenth Symposium on Usable Privacy and Security, SOUPS 2018, Baltimore, MD, USA, August 12-14, 2018.*, pages 297–313, 2018.

[NK16] Sarah Nadi and Stefan Krüger. Variability modeling of cryptographic components: Clafer experience report. In *Proceedings of the Tenth International Workshop on Variability Modelling of Software-intensive Systems, Salvador, Brazil, January 27 - 29, 2016*, pages 105–112, 2016.

[NKMB16] Sarah Nadi, Stefan Krüger, Mira Mezini, and Eric Bodden. Jumping through hoops: why do Java developers struggle with cryptography APIs? In *International Conference on Software Engineering (ICSE)*, pages 935–946, 2016.

[NL08] Nomair A. Naeem and Ondrej Lhoták. Typestate-like analysis of multiple interacting objects. In *Proceedings of the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2008, October 19-23, 2008, Nashville, TN, USA*, pages 347–366, 2008.

[Nor97] David A Northrup. *The problem of the self-report in survey research.* Institute for Social Research, York University, 1997.

[NVLC11] Rick Nunes-Vaz, Steven Lord, and Jolanta Ciuk. A more rigorous framework for security-in-depth. *Journal of Applied Security Research*, 6(3):372–393, 2011.

[NWA+17] Duc-Cuong Nguyen, Dominik Wermke, Yasemin Acar, Michael Backes, Charles Weir, and Sascha Fahl. A stitch in time: Supporting android developers in writing secure code. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 1065–1077, 2017.

[oAIPI] Institute of Applied Information Processing and Communications (IAIK). Automated binary analysis on ios.

[OLR+18] Daniela Seabra Oliveira, Tian Lin, Muhammad Sajidur Rahman, Rad Akefirad, Donovan Ellis, Eliany Perez, Rahul Bobhate, Lois DeLong, Justin Cappos, and Yuriy Brun. API blindspots: Why experienced developers write vulnerable code. In *Fourteenth Symposium on Usable Privacy and Security, SOUPS 2018, Baltimore, MD, USA, August 12-14, 2018*, pages 315–328, 2018.

[Ora19a] Oracle. Java Secure Socket Extension (JSSE) Reference Guide, 2019.

[Ora19b] Oracle. PBEKeySpec Documentation, 2019.

[Ora19c] Oracle. SecureRandom Implementations in JCA Providers, 2019.

[Ora20a] Oracle. SunJCE Provider Documentation, 2020.

[Ora20b] Oracle. SunJSSE Provider Documentation, 2020.

[PFZ17] Olgierd Pieczul, Simon N. Foley, and Mary Ellen Zurko. Developer-centered security and the symmetry of ignorance. In *Proceedings of the 2017 New Security Paradigms Workshop, NSPW 2017, Santa Cruz, CA, USA, October 01-04, 2017*, pages 46–56, 2017.

[PHR19] Nikhil Patnaik, Joseph Hallett, and Awais Rashid. Usability smells: An analysis of developers' struggle with crypto libraries. In *Fifteenth Symposium on Usable Privacy and Security, SOUPS 2019, Santa Clara, CA, USA, August 11-13, 2019*, 2019.

[PSD04] Davide Pozza, Riccardo Sisto, and Luca Durante. Spi2java: Automatic cryptographic protocol java code generation from spi calculus. In *18th International Conference on Advanced Information Networking and Applications (AINA 2004), 29-31 March 2004, Fukuoka, Japan*, pages 400–405, 2004.

[PTRV18] Rumen Paletov, Petar Tsankov, Veselin Raychev, and Martin T. Vechev. Inferring crypto API rules from code changes. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, pages 450–464, 2018.

[PTSA+] Chus Picos, Hoki Torres, Iván García Sáinz-Aja, Steven Coco, Marcos Muíño García, Grégory Joseph, Frank Mena, Carlos Fernández, Éamonn Linehan, Patrick J. Maloney, Soraya Sánchez, Suresh Jaganathan, and Eamonn Dunne. Jasypt: Java simplified encryption. `http://www.jasypt.org/`.

[Pyta] Python Cryptographic Architecture. Cryptography. `https://cryptography.io/en/latest/`.

[Pytb] Python Cryptographic Authority. Pynacl. `https://github.com/pyca/pynacl/`.

[RAMB16] Siegfried Rasthofer, Steven Arzt, Marc Miltenberger, and Eric Bodden. Harvesting runtime values in android applications that feature anti-analysis techniques. In *Network and Distributed System Security Symposium (NDSS)*, February 2016.

[RBK+13] Martin P. Robillard, Eric Bodden, David Kawrykow, Mira Mezini, and Tristan Ratchford. Automated API property inference techniques. *IEEE Transactions on Software Engineering (TSE)*, 39:613–637, 2013.

[REH+16] Michael Reif, Michael Eichberg, Ben Hermann, Johannes Lerch, and Mira Mezini. Call graph construction for java libraries. In *SIGSOFT FSE*, pages 474–486. ACM, 2016.

[Rei03] Frederick F Reichheld. The one number you need to grow. *Harvard Business Review*, 81(12):46–55, 2003.

[Res18] Eric Rescorla. The transport layer security (TLS) protocol version 1.3. *RFC*, 8446:1–160, 2018.

[Rig] RigsIT. Xanitizer.

[RSA78] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, 1978.

[RSJ03] Thomas W. Reps, Stefan Schwoon, and Somesh Jha. Weighted pushdown systems and their application to interprocedural dataflow analysis. In *SAS*, volume 2694 of *Lecture Notes in Computer Science*, pages 189–213. Springer, 2003.

[RXA+19] Sazzadur Rahaman, Ya Xiao, Sharmin Afrose, Fahad Shaon, Ke Tian, Miles Frantz, Murat Kantarcioglu, and Danfeng (Daphne) Yao. Cryptoguard: High precision detection of cryptographic vulnerabilities in massive-sized java projects. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, pages 2455–2472, 2019.

[SAB17] Johannes Späth, Karim Ali, and Eric Bodden. Ide^{al}: Efficient and precise alias-aware dataflow analysis. In *2017 International Conference on Object-Oriented Programming, Languages and Applications (OOPSLA/SPLASH)*. ACM Press, October 2017. To appear.

[SB15] Yannis Smaragdakis and George Balatsouras. Pointer analysis. *Foundations and Trends in Programming Languages*, 2(1):1–69, 2015.

[Sco18] Michael Scovetta. Yasca, 2018.

[SDAB16a] Johannes Späth, Lisa Nguyen Quang Do, Karim Ali, and Eric Bodden. Boomerang: Demand-driven flow- and context-sensitive pointer analysis for java. In *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy*, pages 22:1–22:26, 2016.

[SDAB16b] Johannes Späth, Lisa Nguyen Quang Do, Karim Ali, and Eric Bodden. Boomerang: Demand-driven flow- and context-sensitive pointer analysis for java. In *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy*, pages 22:1–22:26, 2016.

[SDG+14] Shuai Shao, Guowei Dong, Tao Guo, Tianchang Yang, and Chenjie Shi. Modelling analysis and auto-detection of cryptographic misuse in Android applications. In *nternational Conference on Dependable, Autonomic and Secure Computing*, pages 75–80, 2014.

[Smi] Brian Smith. Ring: Safe, fast, small crypto using rust. `https://briansmith.org/rustdoc/ring/`.

[Soda] Sodium. Sodium - a modern, portable, easy to use crypto library. `https://libsodium.org/`.

[Sodb] Sodium Oxide. Sodium oxide: Fast cryptographic library for rust. `https://github.com/sodiumoxide/sodiumoxide`.

[Son17] SonarSource. Sonarqube, 2017.

[Spä19] Johannes Späth. *Synchronized pushdown systems for pointer and data-flow analysis*. PhD thesis, University of Paderborn, Germany, 2019.

[SY86] Robert E. Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Software Eng.*, 12(1):157–171, 1986.

[SZSS19] Larry Singleton, Rui Zhao, Myoungkyu Song, and Harvey P. Siy. Firebugs: finding and repairing bugs with security patterns. In *Proceedings of the 6th International Conference on Mobile Software Engineering and Systems, MOBILESoft@ICSE 2019, Montreal, QC, Canada, May 25, 2019*, pages 30–34, 2019.

[TJVW19] Mohammad Tahaei, Adam Jenkins, Kaim Vaniea, and Maria Wolters. "i don't know too much about it": On the security mindsets of future software creators. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education, Aberdeen, Scotland, UK, July 15-17, 2019*, page 350, 2019.

[TTCL18] Tyler W. Thomas, Madiha Tabassum, Bill Chu, and Heather Lipford. Security during application development: an application security expert perspective. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems, CHI 2018, Montreal, QC, Canada, April 21-26, 2018*, page 262, 2018.

[vdLRWW18] Dirk van der Linden, Awais Rashid, Emma Williams, and Bogdan Warinschi. Safe cryptography for all: towards visual metaphor driven cryptography building blocks. In *Proceedings of the 1st International Workshop on Security Awareness from Design to Deployment, SEAD@ICSE 2018, Gothenburg, Sweden, May 27, 2018*, pages 41–44, 2018.

[Ver17] VeraCode. State of software security 2017. `https://info.veracode.com/report-state-of-software-security.html`, 2017.

[VGH+00] Raja Vallée-Rai, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. Optimizing java bytecode using the soot framework: Is it feasible? In *Compiler Construction*, pages 18–34, 2000.

[WDV+17] Christian Weinert, Denise Demirel, Martín Vigil, Matthias Geihs, and Johannes Buchmann. MoPS: A Modular Protection Scheme for Long-Term Storage. pages 436–448, 2017.

[WLZ+17] Qing Wang, Juanru Li, Yuanyuan Zhang, Hui Wang, Yikun Hu, Bodong Li, and Dawu Gu. Nativespeaker: Identifying crypto misuses in android native code libraries. In *Information Security and Cryptology - 13th International Conference, Inscrypt 2017, Xi'an, China, November 3-5, 2017, Revised Selected Papers*, pages 301–320, 2017.

[WvO08] Glenn Wurster and Paul C. van Oorschot. The developer is the enemy. In *Proceedings of the 2008 Workshop on New Security Paradigms, Lake Tahoe, CA, USA, September 22-25, 2008*, pages 89–97, 2008.

[Xte20] Xtext. XText Website. `http://www.eclipse.org/Xtext/`, 2020.

[ZCD+19] Li Zhang, Jiongyi Chen, Wenrui Diao, Shanqing Guo, Jian Weng, and Kehuan Zhang. Cryptorex: Large-scale analysis of cryptographic misuse in iot devices. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses, RAID 2019, Chaoyang District, Beijing, China, September 23-25, 2019*, pages 151–164, 2019.