

Dealing with Variability in API Misuse Specification

Rodrigo Bonifácio ✉ 

Computer Science Department at University of Brasília

Stefan Krüger ✉

Independent Researcher

Krishna Narasimhan ✉

Technical University of Darmstadt

Eric Bodden ✉ 

Paderborn University & Fraunhofer IEM

Mira Mezini ✉

Technical University of Darmstadt

Abstract

APIs are the primary mechanism for developers to gain access to externally defined services and tools. However, previous research has revealed API misuses that violate the contract of APIs to be prevalent. Such misuses can have harmful consequences, especially in the context of cryptographic libraries. Various API-misuse detectors have been proposed to address this issue—including CogniCrypt, one of the most versatile of such detectors and that uses a language (CrySL) to specify cryptographic API usage contracts. Nonetheless, existing approaches to detect API misuse had not been designed for systematic reuse, ignoring the fact that different versions of a library, different versions of a platform, and different recommendations/guidelines might introduce variability in the correct usage of an API. Yet, little is known about how such variability impacts the specification of the correct API usage. This paper investigates this question by analyzing the impact of various sources of variability on widely used Java cryptographic libraries (including JCA/JCE, Bouncy Castle, and Google Tink). The results of our investigation show that sources of variability like new versions of the API and security standards significantly impact the specifications. We then use the insights gained from our investigation to motivate an extension to the CrySL language (named MetaCrySL), which builds on meta-programming concepts. We evaluate MetaCrySL by specifying usage rules for a family of Android versions and illustrate that MetaCrySL can model all forms of variability we identified and drastically reduce the size of a family of specifications for the correct usage of cryptographic APIs.

2012 ACM Subject Classification Software and its engineering; Software and its engineering → Domain specific languages; Software and its engineering → API languages; Theory of computation → Cryptographic protocols

Keywords and phrases API misuse, cryptographic API misuse detection, code generation, domain engineering, cryptographic standards

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2021.21

Funding This research was supported by the DFG’s collaborative research center 1119 CROSSING and ATHENE National Research Center for Applied Cybersecurity

Rodrigo Bonifácio: funded by FAP-DF (research grant 05/2018)

1 Introduction

Application Programming Interfaces (APIs) have become fundamental to increase developer productivity. Nonetheless, prior research [1, 15, 33] has indicated that developers often struggle with using APIs for various reasons, including poor documentation, low-level abstraction,



© Rodrigo Bonifácio, Stefan Krüger, Krishna Narasimhan, Eric Bodden, Mira Mezini; licensed under Creative Commons License CC-BY 4.0

35th European Conference on Object-Oriented Programming (ECOOP 2021).

Editors: Manu Sridharan and Anders Möller; Article No. 21; pp. 21:1–21:28

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

and lack of tool support. One way to mitigate these issues are approaches to detecting API misuses [2, 12, 20, 25, 27]. Most of these approaches are deny-listing approaches [2, 12, 20, 27]—providing analyses that scan for *incorrect* API uses. Deny-listing approaches suffer from false negatives and cannot be easily extended because they rely on hard-coded rules [25]. To address these issues, CogniCrypt [25] follows an allow-list approach and instead of hard-coding what the correct usages are, it takes a set of correct usage rules as a parameter—the latter are specified by API developers using the CRYSL specification language [25]. For instance, CRYSL has been used to model correct usage rules of Java Cryptographic APIs.

However, the correct usage of an API is often subject to various sources of variability. They include (but are not limited to) evolving signatures and behavioral changes, e.g., due to different security standards in case of crypto APIs.¹ Last but not least, APIs like Java Cryptography Architecture (JCA) foster flexibility through the use of different *providers* that can be *plugged into* to override the default implementation of an algorithm. Depending on the JCA provider, different secure algorithms (according to cryptographic standards) might be available or not. Whether or not an API usage is correct may also vary owing to other factors, including version of the platform (e.g., Java Platform, Android Platform) and version of the API implementations. Nonetheless, there is a lack of understanding about (a) how sources of API variability affect what should be considered the correct usage of an API and (b) a solution to modelling this variability in allow-listing approaches like CRYSL. This is where this paper makes its contributions.

We perform an in-depth domain engineering on the correct usage of cryptographic APIs. To this end, we consider the following sources from which variability might originate from: cryptographic standards (FIPS, ECRYPT, and BSI), cryptographic libraries (e.g., JCA, Google Tink, Bouncy Castle), cryptographic library implementations (e.g., JCA providers), and cryptographic library evolution. Based on the findings, we implement META-CRYSL, a meta-programming approach for managing families of CRYSL specifications, ensuring that different sources of variability can be accounted for when specifying usage patterns. Using the new set of specifications, we conduct an empirical study to investigate two characteristics of META-CRYSL: *expressiveness* (i.e., *the possibility to express all sources of CRYSL variability using META-CRYSL*), *compactness* (i.e., *number of lines of CRYSL code one can save when writing META-CRYSL specifications and the fraction of redundancy one can eliminate*) and *correctness* (i.e., *does the specifications generated by META-CRYSL detect distinct violations when exploring different configurations of CRYSL rules*).

We believe that one can also benefit from using a domain engineering approach for specifying the correct usage of non-cryptographic APIs as well. First because the sources of variability we discuss are not unique to cryptographic APIs as all APIs offer variability in behavior due to evolving signatures as a result of new versions. Second because variability as a result of pluggable implementations from different providers is not unique to JCA, either (c.f., JDBC²). Even security standards that are unique to cryptographic APIs have parallels in the form of context-specific usage patterns for non-crypto APIs.

To summarize, the main contributions of this paper are as follows:

- Domain engineering on Java cryptographic libraries, including:
 - A study on the evolution of Java cryptographic APIs.

¹ Cryptographic libraries have different definitions of correctness—and in particular *secure*—usages, based on the standards like FIPS or BSI under which they operate contributing to yet another source of variability.

² <https://www.oracle.com/database/technologies/jdbc-migration.html>

- A study on different cryptographic standard recommendations.
- A discussion about how the evolution of cryptographic libraries and cryptographic recommendation impact on the correct usage of APIs.
- The design and implementation of META-CRYSL, an extension to CRYSL that helps manage sources of variability on CRYSL specifications.
- An evaluation that shows how META-CRYSL can help API experts to better modularize variability in CRYSL specifications.

In Section 2, we discuss some concepts that are pre-requisite to understanding the remainder of the paper. We present our analysis of sources of API variability in Section 3. In Section 4, we present the design of META-CRYSL, the language that resulted out of the insights gained from our study. Lastly, we empirically evaluate META-CRYSL in Section 5.

2 Background

In this section, we present the concepts and definitions necessary to understand our research context, contributions, and results. In Section 2.1, we introduce the challenges for using cryptographic APIs correctly. Although we perform the first part of our study with different cryptographic APIs, we will use the JCA to drive home these challenges in this section. In Section 2.2, we present the cryptographic standards we consider in our research. These standards may guide and impact the specifications of the correct usage of cryptographic APIs. Finally, Section 2.3 introduces the CRYSL language, which allows experts to specify the proper usage of Java cryptographic APIs.

2.1 Cryptographic APIs

Ferguson et al. [14] state that “cryptography is very difficult”, mostly because it involves several branches of mathematics and computer science [14, 44]. For this reason, algorithms and implementations are only recommended after a huge effort on testing—often conducted by a public community. That is, regardless of how much they have been vetted, they are at best *still secure* or *not yet insecure*. As a result, developers should rely on well-known cryptographic algorithms and API implementations that are subject to hundreds or thousands of hours of cryptanalysis [44].

Cryptographic APIs (or libraries) that exist for each major programming language, such as JCA and Bouncy Castle for Java and wolfCrypt and OpenSSL for C/C++, make available a number of implementations for performing cryptographic tasks, such as the support for generating (pseudo) random numbers, message digests, symmetric and asymmetric cryptography (including digital signature). Although these libraries share similar characteristics, their design differ according to distinct principles, such as flexibility and usability. Unfortunately, existing research reports that these APIs are often complex and hard to use [1, 33], which in the end might compromise the security of the systems.

For instance, JCA has been designed such that it is possible to change the cryptographic implementations used in a system without having to modify many parts of the system. Specifically, this API employs the provider architecture [21] that enables implementations behind the interfaces to be easily swapped. The official documentation of JCA [21] explicitly mentions that the three main motivations driving the design of the API were:

1. **Implementation independence:** Applications can choose between many variants of implementations of cryptographic algorithms

21:4 Dealing with Variability in API Misuse Specification

2. **Implementation interoperability:** Just like the applications are not tied to providers, providers are also not tied to applications
3. **Algorithm extensibility:** Cryptographic algorithms can use building block primitives from variable sources to compose their algorithms

Figure 1 shows a usage scenario for the `MessageDigest` class of the JCA, which computes a hash of input data. The first step to this end is to get an instance of an implementation using a string that specifies the message digest algorithm (`BLAKE2B-512`), and, optionally, a named reference to a provider that makes available the actual implementation of the algorithms through the JCA interface. After getting a `MessageDigest` instance, a developer might populate the digest by calling the `update()` method one or more times, and then calling the `digest` method to compute a hash value of the input data. The same sequence of events has been valid since the first specification of this API. However, several new message digest algorithms have been implemented (e.g., the family of SHA-3 algorithms has been introduced in Java 9). Others have been deprecated and considered insecure over the years (e.g., algorithms MD2, MD5, and SHA-1 are not recommended anymore [16]).

```
@Test
public void testBlakeDigest() {
    try {
        MessageDigest md = MessageDigest.getInstance("BLAKE2B-512", "BC");
        md.update(data);
        byte[] res = md.digest();
        Assert.assertNotNull(res);
    }
    catch(Exception e) {
        org.junit.Assert.fail(e.getMessage());
    }
}
```

■ **Figure 1** Code snippet for computing a message digest using the JCA Bouncy Castle provider (identified by the `BC` string)

Therefore, to correctly use JCA, developers must not only understand the expected sequence of method calls for each cryptographic primitive, but which algorithms and providers are available and are still considered secure. Cryptographic standards detail which algorithms and algorithm configurations developers should use while implementing systems that deal with sensitive information. Given the complexity related to the use of crypto APIs, existing research uses static analysis tools to assess the correct usage of crypto APIs [23, 25, 38] and code generation to assist developers to correctly implement cryptographic tasks [26].

2.2 Cryptographic standards

A cryptographic standard details a set of recommendations related to the use of cryptographic primitives. A few examples of cryptographic standards include:

FIPS Standards present a set of requirements from the American National Institute of Standards and Technology (NIST) that should be considered when implementing security modules for computational systems [34]. This set of standards suggest algorithms for different primitives, including symmetric encryption, digital signatures, and message digest.

BSI TR-021-102-1 is a technical guideline from the German Federal Office for Information Security (BSI) that provides the results of a security assessment on cryptographic

algorithms. This assessment supports a long-term orientation on the use of cryptographic mechanisms [16].

ECrypt TR-D5.4 details a set of recommendations about cryptographic algorithms and key size. It is an effort from the Ecrypt Coordination and Support Action, an initiative from the European Unions' H2020 program [13].

2.3 CrySL: Assessing the Correct Usage of Cryptographic APIs

As can be seen from the above, applying cryptographic APIs within a software can have a lot of potential for errors and developers require new techniques and tools to support the use of cryptographic APIs. A research effort involving different institutions designed and developed CogniCrypt, a suite of tools that leverages the specification language CRYSL to enable API experts to specify the correct usage of libraries. `COGNICRYPTSAST` [25] is a module of CogniCrypt that takes rules in CRYSL and a target program as input and uses state-of-the-art data-flow analysis [45–47] to identify deviations from these rules in this program.

In its current version, CRYSL allows cryptographic experts to specify how to instantiate and use an object-oriented class that implements a cryptographic primitive. Figure 2 shows the CRYSL specification for the `MessageDigest` class of the JCA API.

```

SPEC java.security.MessageDigest
OBJECTS
  java.lang.String algorithm;
  byte[] data;
  byte[] digest;
EVENTS
  g1: getInstance(algorithm);
  g2: getInstance(digestAlg, _);

  Gets := g1 | g2;

  u1: update(_);
  d1: out = digest();
ORDER
  Gets, u1+, d1
CONSTRAINTS
  algorithm in {"SHA-256", "SHA-384", "SHA-512", "BLAKE2B-512"};
ENSURES
  digested[out];

```

■ **Figure 2** CRYSL rule for the `MessageDigest` JCA API (considering the default provider)

A CRYSL rule explicitly states the *class under specification* in the `SPEC` clause. The `OBJECTS` definition describes a list of object declarations. These objects might appear as arguments to events or as variables assigned to the return value of an event. The `EVENTS` section declares the methods of the *class under specification* that are relevant for specifying the correct usage of the class. In particular, the order in which these (labeled) methods should be called appears as a regular expression in the `ORDER` clause. Several operators can be used to denote this regular expression. That is, supposing that we have events with labels e_1 and e_2 , we can combine these events using either the *sequence operator* (e_1, e_2) or the “or” operator ($e_1 | e_2$). We can also state that one event is optional ($e_1?$) or that an event might either occur zero or more times (e_1^*) or one or more times (e_1^+). It is also possible to define *aggregates* (such as `Gets := g1 | g2`), which help with the definition of the `ORDER` clause.

The example of Figure 2 states that a developer must first call one of the `getInstance()` methods (using the `Gets` aggregate) before calling the `update()` method at least once. After that, the developer must conclude the computation of the message digest using the `digest()` method. A CRYSL compiler translates this regular expression into a state machine. After that, the `COGNICRYPTSAST` component [25] analyzes a system to verify if a sequence of calls to a `MessageDigest` instance obeys the expected sequence of events of the `ORDER` clause.

The `CONSTRAINTS` clause allows a cryptographic expert to define constraints on the objects declared in a CRYSL rule. For instance, the CRYSL rule of Figure 2 states that the `algorithm` used as parameter for the `getInstance()` methods should be evaluated to one of the string literals that represent a “secure” message digest algorithm supported by the JCA default providers: `SHA-256`, `SHA-384`, or `SHA-512`. Therefore, during the analysis of a system, `COGNICRYPTSAST` reports an error if it finds a call to the `getInstance()` method of the `MessageDigest` class using a different algorithm (such as `MD5`). Finally, the `ENSURES` clause of a CRYSL rule allows a cryptographic expert to state a predicate that can be later used as a pre-condition in a CRYSL specification for a different class (using the `REQUIRES` construct of CRYSL). There are other CRYSL constructs that we do not discuss here, and a reader that is interested in a more detailed description should read the paper that introduced the CRYSL specification language [25].

Previous studies have shown the efficiency of using the CRYSL approach in identifying common misuses of cryptographic APIs [25], *but considering only one specific set of CRYSL rules*. Nonetheless, as we discuss in the remainder of this paper, CRYSL rules should consider possible sources of variability that might affect the specifications, including versions of APIs and platforms and cryptographic standards.

3 Domain Analysis

To better understand the impacts of variability on API misuse specification, we conducted a *domain analysis* [4, 36] that sought to understand reuse opportunities across Crypto-API usage specifications, considering different libraries, their different providers and their different versions, different cryptographic primitives, and different cryptographic standards—altogether corresponding to the *sources of variability* of our domain analysis. Domain Analysis is a well-established set of activities in the software product line community. The goal is to identify variability motivating the implementation of an infrastructure for software reuse [4, 36].

3.1 Study Settings

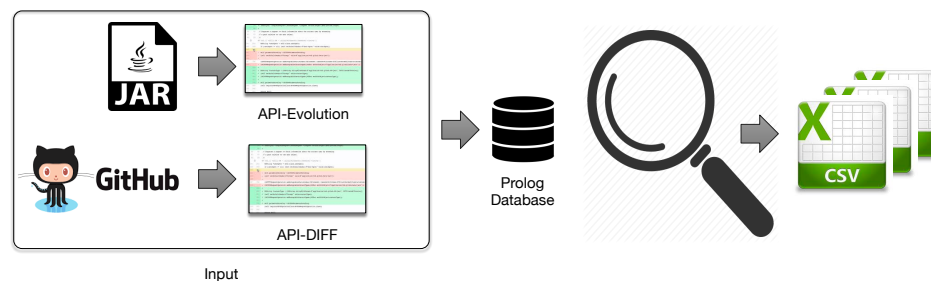
We setup our study based on the following research questions:

- RQ1 How do different APIs and their implementations (e.g., different JCA providers) vary the specifications of the correct usage of cryptographic primitives?** Motivation: Previous studies using specification languages like CrySL only considered the correct usage of the *default* providers for the JCA. These studies report that almost 95% of Android applications that use cryptographic APIs present at least one misuse of these APIs [25]. Answering **RQ1** is relevant because alternative providers such as Bouncy Castle support algorithms that are not supported by the default providers. It is unclear whether findings of the previous CRYSL studies remain valid (particularly in the cases where an application explicitly uses a different provider).
- RQ2 How do existing cryptographic standards vary the notion of secure or compliant use of cryptographic libraries?** Motivation: Although the use of

some cryptographic algorithms are considered insecure (e.g., MD5 and SHA-1), they are still widely used in practice. There are many reasons for that, including compatibility with existing legacy code and the lack of knowledge of developers about up-to-date cryptographic algorithm recommendations. In addition, current security standards (such as FIPS and ECRYPT) present recommendations about which algorithms should be used now and in a near future. Answering **RQ2** helps us to construct a baseline regarding how secure existing applications are when considering existing standards. Moreover, **RQ2** helps to understand the relevance of security standards to the specification of the correct usage of cryptographic APIs.

RQ3 How does the evolution of a cryptographic library vary its correct usage over time? Motivation: Answers to **RQ3** will bring new insights about how to specify policies and static analysis tools that aim to guarantee the correct usage of cryptographic APIs, considering that they might evolve along the way. Moreover, answering **RQ3** might provide evidence that the evolution of APIs must be considered when specifying their correct usage.

To answer the RQs, we first conducted a domain analysis on the specification of the correct usage of Cryptographic APIs. We first read the documentation of APIs and looked at code examples (including test cases) that use Java (JCA, Bouncy Castle, and Google Tink) and C/C++ (OpenSSL and wolfCrypt) cryptographic libraries. We then built a general understanding about how different sources of variability might influence our domain, i.e., the domain of specification of the correct usage of cryptographic APIs.



■ **Figure 3** Approach for mining the evolution of Java Cryptographic APIs

Figure 3 shows the general workflow that we use to mine the evolution of Java cryptographic libraries (JCA, Google Tink, and Bouncy Castle). We leverage APIDIFF [7, 8] and our own static analysis tool to mine classes and methods available per API release and the patterns of changes along the evolution of the libraries. We populate all this information into a Prolog database of facts and rules that allow us to answer questions concerning both newly introduced algorithms as well as deprecated ones for specific versions of a given library. Introducing and removing new primitive algorithms suggest that there should exist CRYSL rules for every version of that API that introduces a change. Using a customized version of APIDIFF, we also investigated breaking changes [6, 8, 49], that is, changes between consecutive releases of an API that break the client code. Next, we execute queries into this database and export the results to CSV files to analyze and understand the evolution of crypto APIs.

```

byte[64] digestData(byte input[64]) {
    byte digest[64];
    Blake2b b2b;
    wc_InitBlake2b(&b2b, 64);
    wc_Blake2bUpdate(&b2b, input, sizeof(input));
    wc_Blake2bFinal(&b2b, digest, 64);
    return digest;
}

```

■ **Figure 4** Function for hashing a byte array using Blake2b

3.2 Analysis Results

RQ1: Variability Due to Different Cryptographic APIs

We started our domain analysis by exploring different cryptographic APIs (e.g., JCA, Google Tink, and Bouncy Castle, Open SSL and wolfCrypt). We soon realized that these APIs differ significantly in terms of design principles and decisions. For instance, the design of the JCA considers flexibility as a key element. Developers are responsible for specifying the configurations of keys and algorithms as well as modes of operations they want to use, which has proven to be challenging to developers. Certain configuration problems might only appear at runtime adding to the complication. The design decisions of Google Tink, on the other hand, favor simplicity, instead of flexibility. This way, there is a small set of key / algorithm configurations available, and the developer is encouraged to use one of these configurations, in order to avoid possible API misuses.

In comparison to Google Tink, C/C++ libraries are yet more restricted. That is, OpenSSL and wolfCrypt define specific functions for each algorithm. The code snippet in Figure 4 shows how to use the wolfCrypt library to generate a hash of an input data using the Blake2b algorithm. There are several calls to functions that are specific to this algorithm. Since the different implementations of message digest algorithms in wolfCrypt do not share a common interface, the code of the `digestData` function is not flexible. In the case a developer has to change the message digest algorithm, she would have to rewrite the entire function.

Based on our analysis of the different APIs, we understand that it is difficult to reuse usage rules across different APIs and languages. Nonetheless, we found some opportunities to reuse CRYSL rules across different JCA providers and within the Google Tink and Bouncy Castle libraries. These opportunities mostly arise due to existing security standard recommendations (we can customize the specifications that address either FIPS or ECRYPT recommendations, for instance), due to the evolution of the API implementations, and due to the similarity we found among different primitives and primitives' implementations. We present some examples of these situations in the remainder of this section.

There is no clear opportunity for reusing the specifications of the correct usage of cryptographic libraries across different APIs and languages.

RQ2: Variability Due to Cryptographic Standards

Existing technical reports and standards present a series of recommendations about which cryptographic algorithms (and respective key configurations) should be used in applications. These technical reports characterize a valuable source of information to indicate whether a given system is “secure according to a given standard”. Moreover, (some) existing

cryptographic APIs (e.g., wolfCrypt and Bouncy Castle) comply to the FIPS certifications—and using a certified library according to the standard recommendations might represent a competitive advantage for products in specific domains. For instance, FIPS 140-2 validation is mandatory for use in the US Federal systems that collect or store sensitive information.³

We found that existing standards introduce a source of variability in usage specifications. This source of variability occurs because sets of algorithms (and algorithm modes) are recommended by some standards, but not in others. Message digests represent one point in variability as Table 1 shows. All standards mentioned in Section 2.2 specify secure hash algorithms that may be used to process a message and produce a condensed representation (a message digest). However, they do not all recommend the same.

Algorithm	FIPS	BSI	ECrypt
MD5	✗	✗	✗
SHA-1	✗	✗	✗
SHA-224	✓	✗	✗
SHA-256, 384, 512	✓	✓	✓
SHA-512/224	✓	✗	✗
SHA-512/256	✓	✓	✓
SHA-3/(256, 384, 512)	✓	✓	✓
Shake128, Shake256	✓	✓	✓
Whirlpool	✗	✗	✓
Blake	✗	✗	✓

■ **Table 1** Recommendations for using different message digest algorithms

If we were to encode these standards in CRYSL, we would need to model them in three distinct rules that nonetheless largely overlap. Let us discuss these rules in more detail. First, consider the default CRYSL specification for the `MessageDigest` class of the JCA, when considering the default provider (Figure 2). In this case, the set of supported algorithms on Line 17 is limited to the default algorithms of JCA.

In case we specify aforementioned standards, we would have to consider using the Bouncy Castle JCA provider—since the default provider does not support some of the algorithms in Table 1 (such as Whirlpool and Blake), and change that line to consider the recommended algorithms of each standard, as we show in Figures 5. In this particular case, it is possible to reuse almost all the CRYSL specification of Figure 2, changing only the `algorithm` constraint based on the supported standard / technical report. We name this kind of variability *Variability on Set Constraints*.

Bouncy Castle provides a lightweight API on top of the providers for JCA⁴. Considering the Lightweight Bouncy Castle API, one is required to write a CRYSL rule for each primitive implementation, as shown in Figure 6 for SHA256 and SHA512. Instead of one specification for each cryptographic standard (varying the supported algorithms), there are several CRYSL rules for each cryptographic standard (one per supported primitive implementation). The variability here relates to the classes that implement the message digest primitives and the Lightweight Bouncy Castle API implements the individual algorithms in a distinct class. However, the corresponding CRYSL specifications vary only according to the base

³ <https://csrc.nist.rip/groups/STM/cmvp/>

⁴ <https://www.bouncycastle.org/>

21:10 Dealing with Variability in API Misuse Specification

```
SPEC java.security.MessageDigest
// same definitions of the default JCA MessageDigest specification
CONSTRAINTS
  algorithm in {"SHA-224", "SHA-256", "SHA-384", "SHA-512", "SHA-3", "Shake-128", "Shake-256"};
ENSURES
  digested[out];
```

(a)

```
SPEC java.security.MessageDigest
// same definitions of the default JCA MessageDigest specification
CONSTRAINTS
  algorithm in {"SHA-256", "SHA-384", "SHA-512", "SHA-3", "Shake-128", "Shake-256"};
ENSURES
  digested[out];
```

(b)

```
SPEC java.security.MessageDigest
// same definitions of the default JCA MessageDigest specification
CONSTRAINTS
  algorithm in {"SHA-256", "SHA-384", "SHA-512", "SHA-3", "Shake-128", "Shake-256", "Whirlpool",
               "Blake2s", "Blake2b"};
ENSURES
  digested[out];
```

(c)

■ **Figure 5** CRYSL rules for the `MessageDigest` JCA (considering the Bouncy Castle provider and the (a) FIPS recommended algorithms, (b) BSI recommended algorithms, and (c) ECRYPT recommended algorithms)

class (in the example, `SHA256Digest` and `SHA512Digest`). We name this kind of variability *Variability on the Base Specification Class*.

The specification of the correct usage of cryptographic APIs should consider the recommendations of individual cryptographic standards. The impact on the specifications due to a cryptographic standard depends on the API.

RQ3: Variability Due to the Evolution of the APIs

We conduct this study using the approach introduced in Section 3.1, to identify cryptographic algorithms introduced/removed and in turn the breaking changes between two public releases of an API. In this case we considered three APIs: JCA, Lightweight Bouncy Castle, and Google Tink. These APIs already have CRYSL specifications for them.

Specifically, we mine the evolution history of 15 releases of the Lightweight Bouncy Castle (v.1.46 to v.1.60), all available in the Maven Central Repository.⁵ Later we summarize some findings related to the evolution of the Google Tink and JCA.

The classes that implement the cryptographic primitives in Lightweight Bouncy Castle implement one of the existing interfaces declared in the Java package `org.bouncycastle.crypto`, including the `Digest`, `Mac`, and `BlockCipher` interfaces. In the last Bouncy Castle release considered in our analysis (release 1.60), we identified more than 140 primitive implementations, among them 45 implementations of the `BlockCipher` interface.⁶ Block cipher (45), message

⁵ <https://search.maven.org/>

⁶ We analyzed these `BlockCipher` implementations and we found classes that implement cipher algorithms

SPEC SHA256Digest	1	SPEC SHA512Digest	1
	2		2
OBJECTS	3	OBJECTS	3
byte input;	4	byte input;	4
byte[] out;	5	byte[] out;	5
int outOff;	6	int outOff;	6
	7		7
EVENTS	8	EVENTS	8
c : SHA256Digest();	9	c : SHA512Digest();	9
u : update(input);	10	u : update(input);	10
f : doFinal(out, outOff);	11	f : doFinal(out, outOff);	11
	12		12
ORDER	13	ORDER	13
c, u+, f	14	c, u+, f	14
	15		15
ENSURES	16	ENSURES	16
digested[out];	17	digested[out];	17

(a)

(b)

■ **Figure 6** Specification of CRYSL rules for the message digest classes in the Bouncy Castle lightweight API. We will have to elaborate one specification for each supported algorithm of a standard / technical report.

digest (29), message authentication code (18), and stream cipher (21) are the primitives with the most algorithm implementations.

Figure 7 shows the evolution in the number of implementations for these primitives. We can see that almost all releases introduce at least one new primitive implementation. For instance, release 1.47 introduced a new implementation of the Mac primitive, while release 1.59 introduced five new block ciphers, one new message digest, and three new stream ciphers. Only releases 1.52, 1.56, and 1.60 did not introduce any new such primitive.

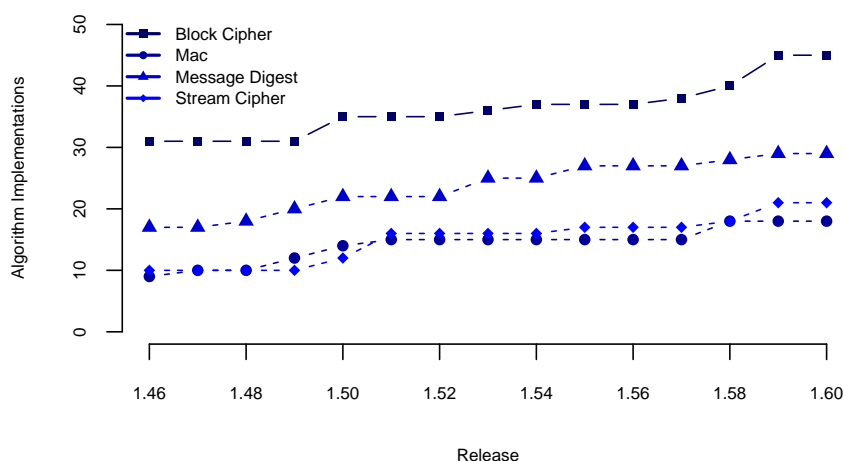
The existence of different implementations of a given primitive has an influence on the specifications of the correct usage of an API. Consider again the test case method on Figure 1. This example uses the Bouncy Castle JCA provider (named “BC”) for generating a digest of an input data using the *Blake2b* algorithm. However, this algorithm was first introduced in the release 1.53 of Bouncy Castle. If one executes this test case using an earlier release (e.g., 1.51 or 1.52), the test case fails with a `NoSuchAlgorithmException`. Therefore, the CRYSL specification of Figure 5(c) is not compliant with the releases of Bouncy Castle prior to 1.53.

What is considered correct usage of an API depends on the specific versions of the API.

We also analyzed the changes in the Bouncy Castle API that might cause an undesired effect on the client systems [49] and identified *breaking changes*. Breaking changes include *removing a public method*, *renaming a public method*, and *reducing visibility of a method*. A catalog of these changes could be found elsewhere [11]. We only consider the twelve releases from 1.49 until 1.60 because these releases are available in the public Git source code repository of Bouncy Castle.

Using the same approach of previous works [7, 8, 49], we found 1.733 scenarios of breaking changes—considering all pairs of successive releases. We document them all in Figure 8. In total, we identified 1.162 removals of public methods (67% of all breaking changes). All

(e.g., AES and Blowfish) and cipher modes (e.g., CBC and GCM).



■ **Figure 7** Evolution in the number of algorithm implementations in Bouncy Castle

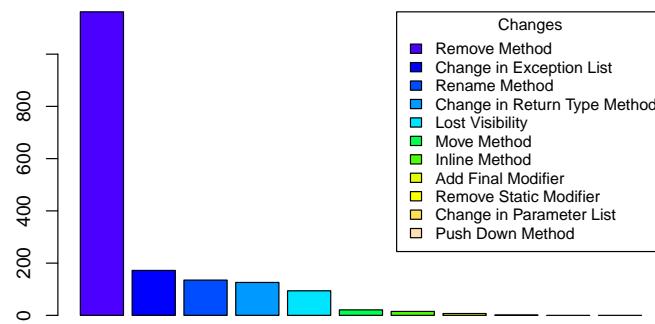
releases feature at least one. Similarly, all releases change the return type of at least one method. There are a total of 128 occurrences. Other common breaking changes are *change in exception list* (172 cases), *renaming a public method* (130 cases), and *reducing visibility of a method* (92 cases). The remaining 49 breaking fall into other categories. Four pairs of successive releases contribute with 74.49% of the breaking changes: 1.58–1.57 (210 cases), 1.57–1.56 (364 cases), 1.51–1.50 (389 cases), and 1.50–1.49 (328 cases). We did not find any evidence that one specific release accounts for a major redesign of the Bouncy Castle library.

Based on these numbers, one might conclude that, despite its long history, Bouncy Castle is an unstable library. This conclusion would be true if developers depended on the public interfaces of the classes that present breaking changes. However, when we consider only the Java interfaces that define the contract of cryptographic primitives (such as the `Digest` and `BlockCipher` interfaces), we found that the Bouncy Castle library is quite stable. Considering all releases, we only identified 34 breaking changes (12 occurrences of *removing a public method*, 9 occurrences of *changing the return type of a public method*, 7 occurrences of *renaming a public method*, 4 occurrences of *reducing visibility of a method*, and 2 occurrences of *changing the exception list of a method*). Yet, we do not know whether or not developers only rely on these “high level” interfaces.

We found 1.733 breaking changes along 11 public releases of Bouncy Castles. However, considering the core interfaces of the library, we only found 34 breaking changes that might also induce changes on CRYSL specifications.

Method updates like renaming/removing/adding methods requires changes to the event section of CRYSL specifications. We name this variability as *Variability on Event Sets*.

We further investigated whether the Google Tink library is more stable: the public interfaces of the classes are almost unchanged between the release 1.0.0 (published in September 2017) and the release 1.2.1 (published in November 2018). During this period, we found 50 breaking changes—43 from version 1.0.0 to version 1.1.0, which might indicate



■ **Figure 8** Total number of breaking changes in the Bouncy Castle API

a slight revision on the first design of the library. Between these first initial releases, we identified 14 removed methods. Nevertheless, the most critical change regarding CRYSL specifications was the introduction of the Deterministic AEAD algorithm on version 1.1.0. This type of variability is modular and involves only the selection of a set of CRYSL specifications (hereafter referred to as *Modular Selection of CRYSL Rule*). Although not common, we also identified some variability due to the key templates available across the different versions of Google Tink. The current CRYSL specifications for Google Tink can deal with the introduction of key templates that modify the events in the specifications (using the *Variability on Event Sets* strategy).

Finally, we also considered the evolution of JCA from Java 4 to Java 9. To this end, we analyzed the classes related to the cryptographic primitives available in three standard Java libraries: `rt.jar`, `jce.jar`, and `sunjce_provider.jar`, considering official releases of the Java language. This API is highly stable as it is based on an official Java specification. For instance, the public class interfaces of the JCA do not present any breaking change, and from Java 5 (2005) to Java 9 (2017) the interface of the `java.security.MessageDigest` class did not change. In Java 7, three additional methods that can be used for ciphering a text with *additional authentication data* (AAD) were introduced in the class `javax.crypto.Cipher`. Although the APIs are stable, new primitive algorithms have been introduced along these versions. For instance, eight new ciphers and six new MAC algorithms have been introduced in the JCA, from Java 4 to Java 9.

4 Meta-CrySL

4.1 Design and Implementation Procedures

We used the outcomes of our domain analysis to design and implement META-CRYSL. META-CRYSL provides means for the systematic reuse of CRYSL specifications. To this end, META-CRYSL allows the specification of CRYSL rules enriched with variation points (such as meta-variables and type parameters) and **refinement** operations that solve these variation points for a given **configuration** (e.g., version of an API or platform, security standard, and so on). META-CRYSL generates a set of CRYSL rules tailored for a given configuration.

We implemented META-CRYSL using Rascal-MPL [24]. One of the main design decisions was to implement three distinct languages: one for abstract CRYSL specifications (i.e., CRYSL with variation points), one for CRYSL refinements, and one for representing a configuration model. The configuration model states a set of abstract CRYSL specifications and refinements. We use a program-generator approach to combine instances of the refinement

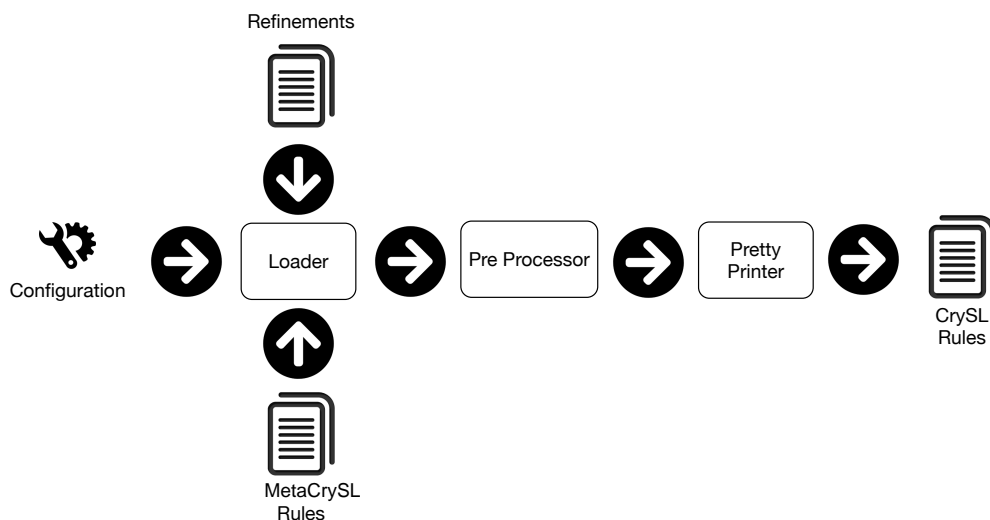
21:14 Dealing with Variability in API Misuse Specification

and configuration languages, and to output regular CRYSL specifications. These regular specifications can directly be used with CogniCrypt’s infrastructure for CRYSL specifications. The following set of high-level requirements guided the design of META-CRYSL.

1. META-CRYSL follows a *meta-programming* approach: we write META-CRYSL specifications and generate regular CRYSL specifications from them. Using this design allows us to preserve all $\text{COGNICRYPT}_{\text{SAST}}$ infrastructure.
2. META-CRYSL should support the sources of variability discussed in the previous section, so that we can generate CRYSL rules for different standards and versions of the APIs.
3. META-CRYSL should also favor reuse among specifications of the same API, reducing the effort in the case that an API supports many algorithms (as for instance Bouncy Castle).

4.2 High-level Architecture

Figure 9 shows the architecture of META-CRYSL, which follows a multi-staged pipeline for language processing [35], where a module loads a META-CRYSL **configuration** that specifies a set of extended CRYSL specifications and a set of **refinements** that should be used during the building process of a specific set of CRYSL rules. After that, the *Loader* module parses the sets of extended CRYSL and refinement files, generating abstract representations of these languages as instances of Rascal-MPL algebraic data types (in the following sections we detail these languages). The *Preprocessor* module manipulates these instances executing the refinement operations, using visitors for program transformations. That is, the *Preprocessor* solves META-CRYSL variability and generates an abstract representation of CRYSL rules. Finally, the *Pretty Printer* module outputs regular CRYSL specification files.



■ **Figure 9** High-level architecture of META-CRYSL

4.3 Abstract CrySL Language

Abstract CRYSL is an extension of the CRYSL language that allows cryptographic specialists to write variation points on the CRYSL rules, for instance, in terms of *meta-variables* and type parameters. Figure 10 shows an example of an instance of the abstract CRYSL language,

modelling variability on CRYSL rules for the JCA `MessageDigest` class. The main source of variability in this case relates to the sets of algorithms that might change due to a specific standard or version of the provider implementation (recall the specifications in Figure 5). The abstract CRYSL rule of Figure 10 introduces the concept of *meta-variables*, which are bound during the derivation process of CRYSL rules. In the example, we can bind the meta-variable `$AlgSet` to the sets of algorithms supported by a given standard (e.g., FIPS, EuroCrypt, or BSI) or specific version of an API.

```

SPEC java.security.MessageDigest
// same definitions of the default JCA MessageDigest specification
CONSTRAINTS
  algorithm in $AlgSet;
ENSURES
  digested[out];

```

■ **Figure 10** Use of meta-variables to deal with *Variability on Set Constraints*

To deal with *Variability on the Base Specification Class*, we use template-based type parameters, similarly to the mechanism of type expansion supported by C++ templates. As such, when solving this type of variability, we actually generate different copies of a CRYSL rule, one for each concrete type that appears in the refinement specifications.

Using the abstract specification of Figure 11, we generated six CRYSL rules for different Google Tink primitive's implementations. We also used the same strategy to factor out existing CRYSL rules for the message digest primitive of the Lightweight Bouncy Castle API. Each one of these CRYSL rules has about 40 lines of code. Using META-CRYSL, we were able to specify each variant using 4 lines of code (three lines of refinements and one line of the configuration language).

```

ABSTRACT SPEC AbstractFactory<T>
OBJECTS
  com.google.crypto.tink.KeysetHandle ksh;
  <T> primitive;
EVENTS
  gp : primitive = getPrimitive(ksh);
ORDER
  gp
REQUIRES
  generatedKeySet[ksh];
ENSURES
  setPrimitive[primitive];

```

■ **Figure 11** Use of type parameters to deal with *Variability on the Base Specification Class*

4.4 Refinement Language

Our refinement language allows cryptographic experts to specify transformations on the META-CRYSL rules, to solve variation points. Considering the discussion of the previous section, META-CRYSL supports two types of syntactic variation points: meta-variables and type parameters. In addition, it is also possible to introduce new events (and events aggregates), to introduce new constraints, and to replace the events' order of a META-CRYSL specification. The refinement language expects a base specification and a list of refinement elements.

21:16 Dealing with Variability in API Misuse Specification

```
SPEC MessageDigest REFINES java.security.MessageDigest {
  define AlgSet = {"Blake2s", "Blake2b", "GOST-3411", "SHA-256", "SHA-384",
                  "SHA-512", "Whirlpool"};
}
SPEC KeyGenerator REFINES javax.crypto.KeyGenerator {
  define AlgSet = {"AES", "BLOWFISH", "HmacSHA256", "HmacSHA384", "HmacSHA512",
                  "RIJNDael", "Serpent"};
  add constraint alg in {"AES"} => keySize in {128, 192, 256};
}
```

■ **Figure 12** Example of refinement specifications for the Bouncy Castle JCA Provider.

```
SPEC SHA256 REFINES
  Digest<org.bouncycastle.crypto.digests.SHA256Digest>;
SPEC SHA384 REFINES
  Digest<org.bouncycastle.crypto.digests.SHA384Digest>;
SPEC SHA512 REFINES
  Digest<org.bouncycastle.crypto.digests.SHA512Digest>;
SPEC SHA512t REFINES
  Digest<org.bouncycastle.crypto.digests.SHA512tDigest>;
```

■ **Figure 13** Example of refinements that bind a type parameter for the set of message digest specifications for the Lightweight Bouncy Castle API.

The current implementation of META-CRYSL supports different refinement transformations, for instance, transformations that support the kinds of variability discussed in the previous section:

- **Define literal set** binds a meta-variable to a literal set, such as `SHA512`, `Blake2b`, `Blake2s`. We use this transformation to solve *Variability on Set Constraints*.
- **Define qualified type** binds a fully qualified type to a type parameter of a META-CRYSL specification. This transformation solves *Variability on the Base Specification Class*.
- **Add new event** introduces a new event into a META-CRYSL specification. We use this transformation to solve *Variability on Event Sets*. Similarly, the refinement language also supports operations to add (remove or update) constraints and requires/ensures clauses.

Figure 12 shows two examples of *refinement specifications*. The first *refines* the `MessageDigest` CRYSL specification of Figure 10, binding the meta-variable `AlgSet` to a set of message digest algorithms supported by the Bouncy Castle JCA provider. The second refinement specification `KeyGenerator` also defines a set of algorithms supported by the Bouncy Castle JCA provider and also introduces a new constraint which refers to two variables of the base specification (not illustrated in this paper): `alg` and `keySize`. The constraint states that if $alg = AES$, the variable `keySize` must be a value in the set $\{128, 192, 256\}$.

Figure 13 shows a set of refinement specifications that are used for generating CRYSL rules for different message digest algorithms supported by the Lightweight Bouncy Castle API. Each refinement specification generates a different CRYSL specification, binding a type parameter with the full qualified name of a class that implements a message digest algorithm. In this scenario, we are able to solve all variability using only type parameters, and thus the body of the refinement specifications is empty.

It might be necessary to add further refinement transformations in the future. To implement a new transformation, one would have to modify three Rascal-MPL modules,


```

1  config android25plus {
2    src = MetaCrySL/samples/jca/base/;
3    out = MetaCrySL/samples/jca/android/target/research/25plus/;
4    load spec base/;
5    load refinement android-bsi/01plus/;
6    load refinement android-bsi/10plus/;
7    load refinement android-bsi/1025/;
8  }

```

■ **Figure 14** Example of a configuration that specifies the rules and refinements target to the version 25 of Android

being necessary to specify the concrete and abstract syntax of the transformation in the refinement language and to implement a new function with the expected behavior of the transformation (Preprocessor module). In case one needs to introduce a new syntactic CRYSL variation point, this is possible by modifying the abstract and concrete syntax of the abstract CRYSL language. We have already implemented six transformations, each one having around ten lines of code.

4.5 Meta-CrySL Configurations

We use a configuration language to specify the META-CRYSL *building process*. Figure 14 shows an example, which states the base path where the META-CRYSL implementation could find the specifications and refinements (Line 2), the output path of the resulting CRYSL specifications (Line 3), and the sets of abstract CRYSL rules and refinements that should be considered during the building process (Lines 4–7). In the example, all CRYSL rules reside in the `base` directory. One may also specify individual rules instead of a directory. The specification of a building process allows cryptographic experts to reuse the same specifications and refinements in different configurations. That is, from the same set of Lightweight Bouncy Castle specifications and refinements, we can create different configurations and generate distinct sets of CRYSL rules. For this flexibility, we opted for such a *configuration language* instead of a *convention-based* mechanism.

5 Empirical Assessment of Meta-CrySL

The **goal** of this empirical assessment is to understand the implications of META-CRYSL in modularizing the specifications of the correct usage of the JCA API for Android, and thereby evaluating META-CRYSL along the lines of *compactness*. Additionally, we also use the empirical assessment to investigate whether or not META-CRYSL generates correct CRYSL specifications, focusing on the *correctness* dimension. Accordingly, we answer the following research **questions** in this empirical assessment, where RQ4 and RQ5 relate to *compactness* and RQ6 explore the *correctness* perspective:

RQ4 How many lines of CRYSL code can one save when writing META-CRYSL specifications?

RQ5 How much duplication of specifications is eliminated by using META-CRYSL in comparison to CRYSL?

RQ6 What are the implications of instantiating CRYSL rules from META-CRYSL specifications, observing the number of API misuses COGNICRYPT_{SAST} analysis reports?

Answering **RQ4** and **RQ5** allows us to quantify the main expected benefit of META-CRYSL: modularizing families of CRYSL specifications with the aim of specification reuse.

Config. Id	Primitives	Android Platform Version	Crypto Standard
C01	All primitives	01 – 08	Android Base recommendations
C02	All primitives	01 – 16	Android Base recommendations
C03	All primitives	01 – 28	Android Base recommendations
C04	All primitives	01 – 08	Android BSI Standard recommendations
C05	All primitives	01 – 16	Android BSI Standard recommendations
C06	All primitives	01 – 28	Android BSI Standard recommendations
C07	All primitives	01 – 08	Android CogniCrypt recommendations
C08	All primitives	01 – 16	Android CogniCrypt recommendations
C09	All primitives	01 – 28	Android CogniCrypt recommendations

■ **Table 2** Sets of cryptographic rules considered in our study

Answering **RQ6** allows us (a) to contrast the difference in the number of reported API misuses when evaluating programs using different META-CRYSL configurations and (b) to check the correctness of our approach for generating CRYSL rules (since COGNICRYPT_{SAST} will reject any invalid CRYSL rule). In this assessment, we used META-CRYSL to modularize a family of CRYSL specifications supporting different **versions** of the Android platform and three **sets of cryptographic recommendations**:

- **Android Base recommendations:** constrains the algorithms that should be used for each version of the Android platform, as detailed in the Android Cryptography Guide specification.⁷
- **Android BSI standard recommendations:** constrains the algorithms considering the BSI standard and the set of Android Base recommendations. The set of Android Base recommendations must be considered because not all BSI recommended algorithms are available in every version of the Android platform.
- **Android CogniCrypt recommendations:** constrains the algorithms according to the current CRYSL specifications from the CogniCrypt project and the set of Android Base recommendations. The set of Android Base recommendations must be considered because not all CogniCrypt recommended algorithms are available in every version of the Android platform.

Specifying the correct usage of the JCA for Android is an interesting scenario for using META-CRYSL, in particular because the correct usage of cryptography in Android depends on the version of the Android platform. Moreover, to answer our research question RQ6, this decision allows us to leverage the same dataset of Android applications that was previously used to empirically assess CRYSL [25]. This dataset comprises 8,136 Android applications, though we could not collect the output of the COGNICRYPT_{SAST} for at least one configuration in a subset comprising 507 of these Android apps. For this reason, we consider a smaller set of 7,629 Android apps. From our META-CRYSL specifications, we can generate hundreds of configurations. Since it is computationally expensive to run COGNICRYPT_{SAST} on a dataset with thousands of Android apps, we decided to conduct our assessment with the nine configurations shown in Table 2. Each configuration supports all cryptographic primitives (JCA supports 32 primitives in total, including Block Cipher and Message Digest), one of three distinct ranges of versions of the Android platform (01 – 08, 01 – 16, 01 – 28), and one of the cryptographic recommendations.

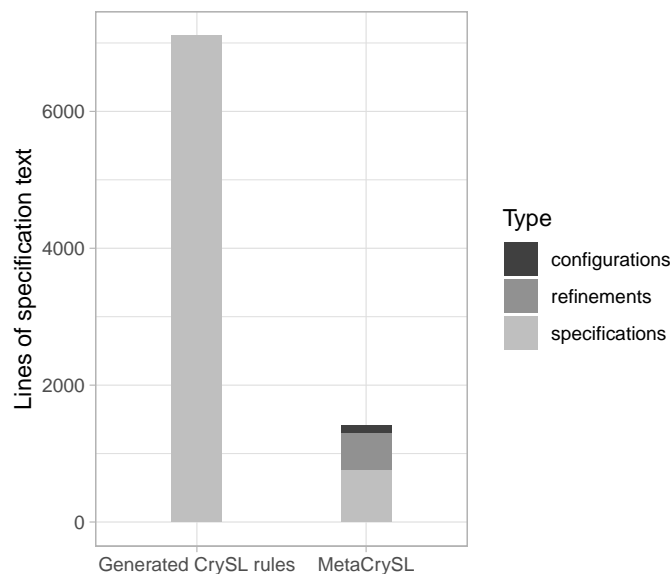
We answer research questions **RQ4**, **RQ5**, and **RQ6** through the use of metrics. For **RQ4** we compute (a) the total number of lines in META-CRYSL necessary to specify the

⁷ Android Cryptography Guide: <https://developer.android.com/guide/topics/security/cryptography>

sets of configurations of Table 2 and (b) the resulting lines of specifications in CRYSL that we generate using the META-CRYSL specifications. We then compute how many lines of specification text we save using META-CRYSL. For **RQ5** we estimate the total number of duplication in the META-CRYSL specifications, as well as in the generated CRYSL rules. We answer **RQ6** using the *total number of rule violations* that `COGNICRYPTSAST` finds in the dataset of Android applications when using each distinct set of CRYSL rule configurations.

5.1 RQ4: How many lines of CrySL code can one save when writing Meta-CrySL specifications?

In **RQ4**, we investigate the benefits of using META-CRYSL w.r.t. removing the redundant code that one would write when specifying the sets of CRYSL rules describing the correct usage of cryptographic APIs—tailored to the nine configurations in Table 2. Figure 15 summarizes the total number of lines needed to write the META-CRYSL specifications, refinements, and configurations as well as the total number of lines of specifications generated by META-CRYSL and that could be used to execute `COGNICRYPTSAST` with the distinct configurations. In this case study, we wrote 1,407 lines in META-CRYSL (762 lines of META-CRYSL specifications, 540 lines of META-CRYSL refinements, and 105 lines of META-CRYSL configurations), and generated 7,105 lines of CRYSL rules for those configurations, saving 80% of lines.

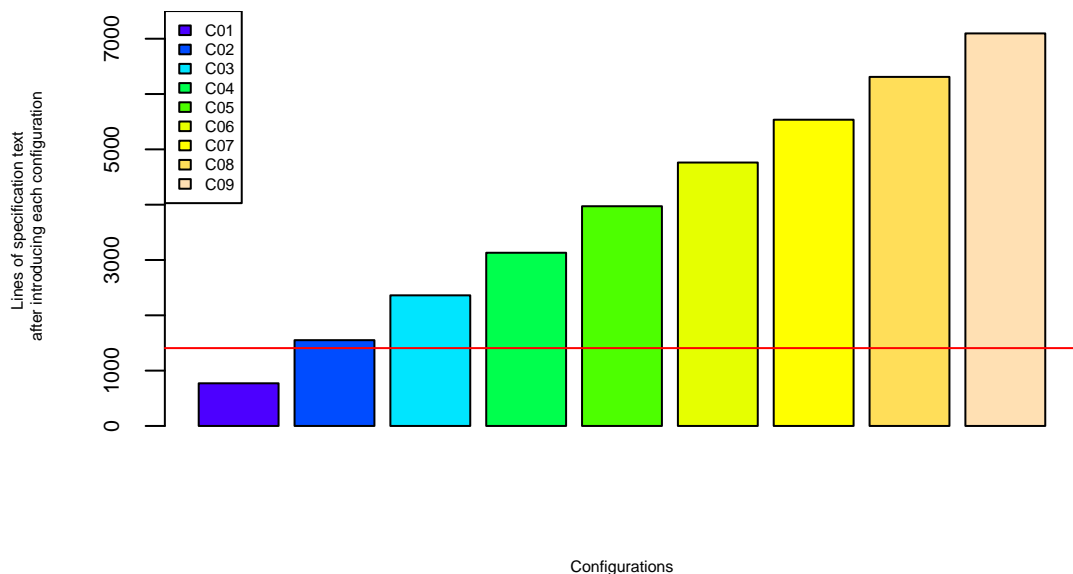


■ **Figure 15** The total number of lines of code necessary to specify the nine configurations in META-CRYSL (including META-CRYSL specifications, META-CRYSL refinements, and META-CRYSL configurations) and the total number of lines of CRYSL code generated.

META-CRYSL removed 80% of the redundancy induced when writing all the CRYSL rules tailored to the specific configurations considered in our study.

The META-CRYSL payoff tends to increase when defining new configurations, since one would then generate further instances of CRYSL from the same set of META-CRYSL rules

and refinements. Figure 16 shows how many lines of CRYSL specification we generate after introducing each configuration in Table 2. In terms of lines of specification text, we achieve a payoff after generating the second configuration (C02).



■ **Figure 16** Evolution of the total lines of generated CRYSL specification text after introducing each configuration. The red line corresponds to the total number of lines of META-CRYSL used to generate the configurations

5.2 RQ5: How much duplication of specifications is eliminated by using Meta-CrySL in comparison to CrySL?

In total, there were 188 files (including refinements and configurations) of base META-CRYSL specifications for the JCA use in Android. These files contained 1407 lines of specifications, out of which 633 lines were duplicates, resulting out of 156 individual lines. In comparison, the corresponding CRYSL specifications for three families of Android configurations (BSI, CogniCrypt, Base) each comprising specifications for three versions (0108, 0116, 25plus) contributed to 7,105 lines of specifications spread across 288 files. Out of these, 5,579 lines of specifications were duplicates resulting out of 546 unique lines.

The amount of duplicate lines of specifications for a family of CRYSL specifications is 5,579 in comparison to 633 for META-CRYSL specifications for the same family (11.34%).

Most of the duplication in META-CRYSL arises because we specified all META-CRYSL refinements for the three families of Android configurations (BSI, CogniCrypt, and Base) which could be prevented by writing carefully crafted refinements. Specifically, out of the 1407 lines of specifications, only 97 lines and 85 lines of duplicates resulted from the base META-CRYSL specifications and configurations— 451 of the 633 duplicates resulted from refinements for individual versions.

5.3 RQ6: What are the implications of instantiating CrySL rules from Meta-CrySL specifications, observing the number of API misuses CogniCrypt_{sast} analysis reports?

Our research question RQ6 explores the results of COGNICRYPT_{SAST} for the nine configurations (C01 – C09). We concentrate our analysis on the violations related to the CONSTRAINTS section of CRYSL rules, mostly because cryptographic standards do not address other sections. Table 3 summarizes the results of the analysis, showing the number of Android apps using the JCA APIs, the number of Android apps using the JCA APIs incorrectly (i.e., presenting at least one misuse), the rate of vulnerable Android apps (calculated using the previous two), and the total number of violations. The results of COGNICRYPT_{SAST} reveal a significant number of apps with at least one JCA API misuse in all configurations—more than five percent of the apps present at least one misuse in the more permissive Android Base sets of recommendations. This number jumps to more than 45% when considering the BSI or the CogniCrypt recommendations.

Configuration	Apps Using JCA	Apps Presenting Misuse	(Rate %)	# Violations
Android Base 0108	6,714	545	8.12	1,083
Android Base 0116	6,714	395	5.88	1,224
Android Base 25plus	6,714	386	5.75	830
Android BSI 0108	6,714	3,184	47.42	9,089
Android BSI 0116	6,714	3,155	46.99	8,905
Android BSI 25plus	6,714	3,155	46.99	8,873
Android CogniCrypt 0108	6,714	3,261	48.57	9,077
Android CogniCrypt 0116	6,714	3,260	48.56	8,975
Android CogniCrypt 25plus	6,714	3,256	48.50	8,945

■ **Table 3** Summary of the findings of COGNICRYPT_{SAST} for the different CrySL configurations.

The total number of violations when considering the set of Android Base recommendations is substantially smaller than the total number of violation found using the other configurations (Android-BSI and Android-CC configurations) and most of the violations in the Android Base configurations relate to the *Cipher* primitive. For instance, when one only considers the “Android Base 25plus” configuration, 548 out of 830 violations are either due to the use of an insecure cipher algorithm (such as DES or DESede) or due to the use of an insecure algorithm/mode/padding configuration (e.g., AES/CBC/NoPadding). This changes when one considers the other sets of recommendations (from Android BSI and Android CogniCrypt). There, most of the violations relate to the *Message Digest* primitive. For instance, considering the Android BSI 0116 configuration, one finds 6,272 violations due to insecure message-digest algorithms (e.g., MD5 and SHA-1)—this corresponds to 70.43% of all violations one finds with this particular configuration.

Regarding the differences between the Android BSI and Android CogniCrypt families of CRYSL rules we found some modes of operations that are not mentioned in the BSI standard (e.g., RSA/ECB/PKCS1Padding) but that are considered secure and recommended in CogniCrypt. The *Message Authentication Code* (MAC) primitive also brings differences in the number of violations when comparing the BSI and CogniCrypt recommendations. Actually, the BSI standard makes clear that the HMAC scheme should only be used with the SHA-2 or SHA-3 families of hash functions, though the algorithms HmacMD5 and HmacSHA1 are allowed by the CogniCrypt configurations.

21:22 Dealing with Variability in API Misuse Specification

We also found some differences when considering the particular platform versions. For instance, until version 10 of the Android platform, developers must use the TLS⁸ algorithm for `SSLContext`. This led to 169 additional violations regarding the incorrect usage of the `SSLContext` class in the “Android Base 0108” configuration, in comparison to “Android Base 0116” and “Android Base 25plus”. In more detail, 161 apps use either the SSL or TLSv1 algorithms (both introduced in version 10) and eight apps use either TLSv1.1 or TLSv1.2 (both introduced in version 10). These violations do not occur in the remaining “0116” and “25plus” configurations. We also found similar divergences on the platform version related to other cryptographic primitives.

It is important to note that, although version 8 was released in May 2010 already, in order to increase compatibility with a broader range of devices, most apps in our dataset are still configured to use this version as the minimum version. The observation that the number of violations for the “Android Base 0108” configuration is higher compared to the the “Android Base 0116” and “25plus” configurations might indicate that some apps use cryptographic algorithms that are not available in their minimum version. This would then lead to a runtime exception. In summary:

The experiments showed a significant difference when considering the different versions of the platform for the Android Base configurations. Yet, the Android Base configurations are much less restrictive than those of the BSI and by CogniCrypt in general. We found slight differences in the results of `COGNICRYPTSAST` when considering the recommendations from BSI and CogniCrypt. Although the differences are not that large, this result still suggests that one can benefit from tailoring the specifications of the correct usage of cryptographic standards according to the different guidelines.

6 Threats to Validity

In this section, we present some limitations and possible threats to the validity of our work. Since our research focuses on cryptographic libraries only, we need to discuss the applicability of our approach to other domains. The choice of this domain was motivated by our previous experience using `CRYSL` to specify the correct usage of cryptographic APIs. We was challenged by the fact that new algorithms are frequently designed and old ones might become deprecated [5]. In addition, cryptographic standards are frequently updated—in particular to state that an algorithm vulnerability has been found and reported.

We believe that our approach can also be used for APIs that target other domains as well, even though we did not systematically investigate this question. First, APIs from different domains evolve along the time, and as we discussed throughout this paper, API evolution has an impact on the correct usage of libraries. Second, there are different recommendations on the proper usage of each popular API. For instance, there are many guidelines discussing the correct usage of the Java Persistence API [?, ?]—and individual companies might also take advantage of specific recommendations. The specifications about how to correctly use a given API should take into account these differences. We envision that both practitioners and researchers would benefit from a domain engineering approach that considers different sources of variability— including different versions of an API, recommendations from gray literature (for instance), and mining software repositories efforts—before specifying the correct usage a given API. We make the reader aware that domain engineering is a well-known technique to

⁸ TLS is a protocol that provides authenticated encryption for data connections.

understand properties that, like in our case, bring variability to the domain of API usage specifications. We are not attempting to validate domain engineering itself or propose a technique for its application to other domains; the process for which would require careful understanding of the specific API domain and a thorough analysis.

Another threat to our conclusions relates to the additional complexity introduced by META-CRYSL. We envision that the users of META-CRYSL are already users of CrySL, and the learning curve would involve a language for specifying CRYSL refinements and configurations. To better quantify the additional complexity META-CRYSL introduces, we will have to conduct a *user study* with this specific goal. We postpone such an investigation to a future work, since our focus here was to explore META-CRYSL in a more realistic scenario, investigating the possible benefits of using META-CRYSL to modularize CRYSL specifications for different versions of the Android platform and different cryptographic recommendations. Therefore, currently we do not have empirical evidence about how much complexity META-CRYSL introduces to those already familiar with CRYSL. Nonetheless, compared to the benefit of managing a large family of specifications using a relatively small number of refinements and configurations, we feel this additional complexity is justified.

Additional threats relate to the methods we used in our research. We tried to mitigate possible *reliability threats* by reusing methods and tools from previous research studies. For instance, we investigated the frequency of *breaking changes* to estimate the stability of Java cryptographic libraries using the methodologies available in the literature [7, 8, 49]. Nonetheless, although we found more than 1700 *breaking changes* across 11 public releases of Bouncy Castle, a limitation of our work is that we do not investigate if these changes have actually broken existing client code. Our understanding is that just a subset of breaking changes impact on the specifications of the correct usage of APIs.

This threat relates to the use of APIDIFF, which detects breaking changes considering modifications to the *standard notion* of Java interfaces—that is, public members of Java classes or interfaces. Modifications that do not preserve the standard notion of Java interfaces (e.g., changing the signature of public methods, removing public methods, and so on) are claimed by APIDIFF as breaking changes. This might actually lead to a number of false-positives—once client code might not depend on all public members of a library. To mitigate this threat, we narrowed our analysis of the Bouncy Castle library to only focus on the high-level classes and interfaces of Bouncy Castle that implement cryptographic primitives.

Regarding our research question **RQ4**, we measure the reduction of lines of specification and redundancy with respect to *generated* specifications. This might raise the question whether these generated specifications do not contain boilerplate text that had not arisen had these specifications be hand-written. We are confident that we can rule this out, due to the nature of CRYSL specifications and the way they are generated by META-CRYSL. Conducting a large scale developer study by manually writing many families of specifications by hand was beyond the scope of this work.

7 Related Work

7.1 Domain Engineering

Frakes et al. [17] present a well-established definition for domain engineering, which embraces two phases: *domain analysis* and *domain implementation*. The first deals with all activities necessary to understand and document the commonalities and variabilities within a software domain. Similar to the guidelines presented by the authors, we also collected and recorded information from documents (cryptographic standards) and source code (examples of

cryptographic libraries usage) while conducting our domain analysis. The main difference of our approach is that we also mined the source code evolution of the cryptographic libraries. Lisboa et al. presents a literature review on tools and methods for domain analysis [28].

The second phase of domain engineering (that is, domain implementation) aims to build the infrastructure necessary to generate products from reusable assets. Here we used the same general idea, though not to build software products, but actually to generate specifications of the correct usage of APIs that might vary according to different sources of variability (such as versions of APIs, platforms, and cryptographic standards). Czarnecki and Eisenecker [10] detail several techniques that can be used to implement an infrastructure for building products from reusable assets. In our work, we used the *refinement-based* transformational approach [10, Chapter 9] as the basis for the META-CRYSL design and implementation. The literature on software product lines also recommends two distinct phases for building SPLs: one for domain analysis and one for domain implementation [4, 36].

7.2 Correct Usage of APIs

Amann et al. present some terminology and taxonomy around the correct usage and misuse of APIs [3]. Given a set of constraints stating, for instance, the expected order of method calls and the pre-conditions the client code must guarantee before calling the methods of an API, any usage scenario that violates a constraint characterizes a misuse—otherwise, it is a correct usage [3]. The main goal of mining misuse of APIs is to reveal *deviant code* that might originate a bug or a software vulnerability (in the context of cryptographic APIs).

According to Amann et al [3], the constraint specifications could be manually crafted by experts or inferred using either dynamic [30, 37] or static analysis [32, 40, 48]. In this paper, we rely on a manually crafted approach to specify rules in META-CRYSL—mostly because many programs fail to use cryptographic APIs correctly [1, 25, 33]. It is a matter of future work to investigate if our domain engineering approach could also benefit from techniques that automatically infer the correct usage of APIs.

To the best of our knowledge, none of the previous research works consider that the correct usage of an API could vary, among other reasons, according to specific versions of APIs or to existing usage recommendation patterns that could be general accepted or tailored to particular companies or projects.

7.3 API Evolution

Studies on API evolution focus on two directions. First, to help developers to migrate their systems in response to the evolution of APIs the systems depend on [9, 18, 31, 43]. The second direction, which is closely related to our research, focus on understanding how developers evolve APIs and on characterizing the evolution of APIs. For instance, several research works have explored the impact of *deprecation* mechanisms on software ecosystems [39, 41, 42]. Other research studies investigate how developers respond to API evolution [19] and the motivations for breaking APIs [6].

Here we investigate how the evolution of cryptographic APIs occurs in practice, considering the history of three Java cryptographic libraries: JCA/JCE, Bouncy Castle, and Google Tink. We have found that cryptographic libraries are quite stable, and the high-level APIs that define cryptographic primitives rarely change—even though we found a number of *breaking changes* during the evolution of Bouncy Castle. The most typical pattern is the introduction of new algorithms that implement cryptographic primitives—which often requires changes into the specification about the correct usage of the APIs.

8 Conclusion

Domain engineering involves a set of techniques for identifying and documenting the commonalities and variabilities within a software domain, as well as for building an infrastructure for deriving products from reusable assets [4, 17, 36]. While it has been successfully used to develop software product lines, in this paper, we explored the use of domain engineering procedures to specify the correct usage of cryptographic APIs. After gathering a better understanding about how different versions of the platforms, APIs, and cryptographic standards might affect the specifications of the correct usages of crypto APIs, we designed META-CRYSL. META-CRYSL serves as an infrastructure for generating CRYSL [25] specifications tailored for specific scenarios. We evaluated our approach using a family of META-CRYSL specifications describing the correct usage of the Java Cryptographic Architecture for Android, which accommodates the evolution of the Android platform and three distinct sets of cryptographic recommendations. Our results provide evidence that it is important to tackle the problem of writing specifications of correct usage of APIs using a domain engineering approach and that using META-CRYSL we can better modularize families of specifications.

References

- 1 Y. Acar, M. Backes, S. Fahl, S. Garfinkel, D. Kim, M. L. Mazurek, and C. Stransky. Comparing the usability of cryptographic apis. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 154–171. IEEE Press, May 2017. doi:10.1109/SP.2017.52.
- 2 S. Amann, H. A. Nguyen, S. Nadi, T. N. Nguyen, and M. Mezini. A systematic evaluation of static api-misuse detectors. *IEEE Transactions on Software Engineering*, 45(12):1170–1188, 2019.
- 3 S. Amann, H. A. Nguyen, S. Nadi, T. N. Nguyen, and M. Mezini. A systematic evaluation of static api-misuse detectors. *IEEE Transactions on Software Engineering*, 45(12):1170–1188, 2019. doi:10.1109/TSE.2018.2827384.
- 4 Sven Apel, Don Batory, Christian Kstner, and Gunter Saake. *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer Publishing Company, Incorporated, 2013.
- 5 A. Bhardwaj and S. Som. Study of different cryptographic technique and challenges in future. In *2016 International Conference on Innovation and Challenges in Cyber Security (ICICCS-INBUSH)*, pages 208–212, 2016.
- 6 Aline Brito, Marco Tulio Valente, Laerte Xavier, and André C. Hora. You broke my code: understanding the motivations for breaking changes in apis. *Empirical Software Engineering*, 25(2):1458–1492, 2020. doi:10.1007/s10664-019-09756-z.
- 7 Aline Brito, Laerte Xavier, André C. Hora, and Marco Tulio Valente. Apidiff: Detecting API breaking changes. In Rocco Oliveto, Massimiliano Di Penta, and David C. Shepherd, editors, *25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018*, pages 507–511. IEEE Computer Society, 2018. doi:10.1109/SANER.2018.8330249.
- 8 Aline Brito, Laerte Xavier, André C. Hora, and Marco Tulio Valente. Why and how Java developers break APIs. In Rocco Oliveto, Massimiliano Di Penta, and David C. Shepherd, editors, *25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018*, pages 255–265. IEEE Computer Society, 2018. doi:10.1109/SANER.2018.8330214.
- 9 Kingsum Chow and David Notkin. Semi-automatic update of applications in response to library changes. In *1996 International Conference on Software Maintenance (ICSM '96), 4-8 November 1996, Monterey, CA, USA, Proceedings*, page 359. IEEE Computer Society, 1996. doi:10.1109/ICSM.1996.565039.

- 10 Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. ACM Press/Addison-Wesley Publishing Co., USA, 2000.
- 11 Danny Dig and Ralph Johnson. How do apis evolve? a story of refactoring: Research articles. *J. Softw. Maint. Evol.*, 18(2):83–107, March 2006.
- 12 Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. An empirical study of cryptographic misuse in android applications. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security, CCS '13*, pages 73–84, New York, NY, USA, 2013. ACM. URL: <http://doi.acm.org/10.1145/2508859.2516693>, doi: 10.1145/2508859.2516693.
- 13 Michel Abdalla et al. Algorithms, key size and protocols report. Technical report, ECRYPT – Coordination and Support Action, European Union’s H2020 programme, 2018.
- 14 Niels Ferguson, Bruce Schneier, and Tadayoshi Kohno. *Cryptography Engineering: Design Principles and Practical Applications*. Wiley Publishing, 2010.
- 15 F. Fischer, K. Böttinger, H. Xiao, C. Stransky, Y. Acar, M. Backes, and S. Fahl. Stack overflow considered harmful? the impact of copy amp;paste on android application security. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 121–136, May 2017. doi: 10.1109/SP.2017.31.
- 16 German Federal Office for Information Security. Cryptographic mechanisms: Recommendations and key lengths. Technical Report BSI TR-02102-1, German Federal Office for Information Security, 2020.
- 17 William Frakes, Ruben Prieto, Christopher Fox, et al. Dare: Domain analysis and reuse environment. *Annals of software engineering*, 5(1):125–141, 1998.
- 18 Johannes Henkel and Amer Diwan. Catchup! capturing and replaying refactorings to support api evolution. In *Proceedings of the 27th International Conference on Software Engineering, ICSE '05*, page 274–283, New York, NY, USA, 2005. Association for Computing Machinery. doi:10.1145/1062455.1062512.
- 19 A. Hora, R. Robbes, N. Anquetil, A. Etien, S. Ducasse, and M. Tulio Valente. How do developers react to api evolution? the pharo ecosystem case. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 251–260, 2015. doi: 10.1109/ICSM.2015.7332471.
- 20 David Hovemeyer and William Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12):92–106, December 2004. doi:10.1145/1052883.1052895.
- 21 Oracle Inc. Java cryptography architecture (JCA), 2020. <https://docs.oracle.com/en/java/javase/15/security/java-cryptography-architecture-jca-reference-guide.html>.
- 22 Gökhan Kahraman and Semih Bilgen. A framework for qualitative assessment of domain-specific languages. *Software & Systems Modeling*, 14(4):1505–1526, Oct 2015. doi:10.1007/s10270-013-0387-8.
- 23 Maria Kechagia, Xavier Devroey, Annibale Panichella, Georgios Gousios, and Arie van Deursen. Effective and efficient api misuse detection via exception propagation and search-based testing. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019*, page 192–203, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3293882.3330552.
- 24 Paul Klint, Tijs van der Storm, and Jurgen J. Vinju. RASCAL: A domain specific language for source code analysis and manipulation. In *Ninth IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2009, Edmonton, Alberta, Canada, September 20-21, 2009*, pages 168–177. IEEE Computer Society, 2009. doi:10.1109/SCAM.2009.28.
- 25 S. Krüger, J. Späth, K. Ali, E. Bodden, and M. Mezini. Crysl: An extensible approach to validating the correct usage of cryptographic apis. *IEEE Transactions on Software Engineering*, pages 1–1, 2019. doi:10.1109/TSE.2019.2948910.
- 26 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, pages 931–936. IEEE Press, 2017.
- 27 Yong Li, Yuanyuan Zhang, Juanru Li, and Dawu Gu. icryptotracer: Dynamic analysis on misuse of cryptography functions in ios applications. In Man Ho Au, Barbara Carminati, and C.-C. Jay Kuo, editors, *Network and System Security*, pages 349–362, Cham, 2014. Springer International Publishing.
 - 28 Liana Barachisio Lisboa, Vinicius Cardoso Garcia, Daniel Lucrédio, Eduardo Santana de Almeida, Silvio Romero de Lemos Meira, and Renata Pontin de Mattos Fortes. A systematic review of domain analysis tools. *Information and Software Technology*, 52(1):1 – 13, 2010. URL: <http://www.sciencedirect.com/science/article/pii/S0950584909000834>, doi:<https://doi.org/10.1016/j.infsof.2009.05.001>.
 - 29 Cristina Videira Lopes. *Exercises in Programming Style*. Chapman & Hall/CRC, 2014.
 - 30 Qingzhou Luo, Yi Zhang, Choonghwan Lee, Dongyun Jin, Patrick O’Neil Meredith, Traian Florin Șerbănuță, and Grigore Roșu. Rv-monitor: Efficient parametric runtime verification with simultaneous properties. In Borzoo Bonakdarpour and Scott A. Smolka, editors, *Runtime Verification*, pages 285–300, Cham, 2014. Springer International Publishing.
 - 31 Mira Mezini. Maintaining the consistency of class libraries during their evolution. *SIGPLAN Not.*, 32(10):1–21, October 1997. doi:10.1145/263700.263701.
 - 32 Martin Monperrus and Mira Mezini. Detecting missing method calls as violations of the majority rule. *ACM Trans. Softw. Eng. Methodol.*, 22(1), March 2013. doi:10.1145/2430536.2430541.
 - 33 Sarah Nadi, Stefan Krüger, Mira Mezini, and Eric Bodden. Jumping through hoops: Why do java developers struggle with cryptography apis? In *Proceedings of the 38th International Conference on Software Engineering*, ICSE ’16, pages 935–946. ACM, 2016. doi:10.1145/2884781.2884790.
 - 34 National Institute of Standards and Technology. Security requirements for cryptographic modules. Technical report, National Institute of Standards and Technology, 2019.
 - 35 Terence Parr. *Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages*. Pragmatic Bookshelf, 1st edition, 2009.
 - 36 Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag, Berlin, Heidelberg, 2005.
 - 37 Michael Pradel and Thomas R. Gross. Leveraging test generation and specification mining for automated bug detection without false positives. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE ’12, page 288–298. IEEE Press, 2012.
 - 38 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 ’19, page 2455–2472, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3319535.3345659.
 - 39 Romain Robbes, Mircea Lungu, and David Röthlisberger. How do developers react to api deprecation? the case of a smalltalk ecosystem. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE ’12, New York, NY, USA, 2012. Association for Computing Machinery. doi:10.1145/2393596.2393662.
 - 40 M. A. Saied, O. Benomar, H. Abdeen, and H. Sahraoui. Mining multi-level api usage patterns. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 23–32, 2015. doi:10.1109/SANER.2015.7081812.
 - 41 A. A. Sawant, R. Robbes, and A. Bacchelli. On the reaction to deprecation of 25,357 clients of 4+1 popular java apis. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 400–410, 2016. doi:10.1109/ICSME.2016.64.
 - 42 Anand Ashok Sawant, Romain Robbes, and Alberto Bacchelli. On the reaction to deprecation of clients of 4+ 1 popular java apis and the jdk. *Empirical Software Engineering*, 23(4):2158–2197, 2018.

- 43 Thorsten Schäfer, Jan Jonas, and Mira Mezini. Mining framework usage changes from instantiation code. In *Proceedings of the 30th International Conference on Software Engineering, ICSE '08*, page 471–480, New York, NY, USA, 2008. Association for Computing Machinery. doi:10.1145/1368088.1368153.
- 44 Bruce Schneier. *Secrets & Lies: Digital Security in a Networked World*. John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 2000.
- 45 Johannes Späth, Karim Ali, and Eric Bodden. Ide^{al}: efficient and precise alias-aware dataflow analysis. *PACMPL*, 1(OOPSLA):99:1–99:27, 2017. doi:10.1145/3133923.
- 46 Johannes Späth, Karim Ali, and Eric Bodden. Context-, flow-, and field-sensitive data-flow analysis using synchronized pushdown systems. *PACMPL*, 3(POPL):48:1–48:29, 2019.
- 47 Johannes Späth, Lisa Nguyen Quang Do, Karim Ali, and Eric Bodden. Boomerang: Demand-driven flow- and context-sensitive pointer analysis for java. In Shriram Krishnamurthi and Benjamin S. Lerner, editors, *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy*, volume 56 of *LIPICs*, pages 22:1–22:26. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016. doi:10.4230/LIPICs.ECOOP.2016.22.
- 48 Andrzej Wasylkowski, Andreas Zeller, and Christian Lindig. Detecting object usage anomalies. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, ESEC-FSE '07*, page 35–44, New York, NY, USA, 2007. Association for Computing Machinery. doi:10.1145/1287624.1287632.
- 49 L. Xavier, A. Brito, A. Hora, and M. T. Valente. Historical and impact analysis of api breaking changes: A large-scale study. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 138–147, Feb 2017. doi:10.1109/SANER.2017.7884616.