UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft
Faculty for Computer Science, Electrical Engineering and Mathematics
Department of Computer Science
Research Group Software Engineering

Bachelor's Thesis

Submitted to the Software Engineering Research Group
in Partial Fullfilment of the Requirements for the Degree of

Bachelor of Science

Testing and Evaluation of CogniCrypt and CrySL in the Context of non-cryptography-related API Misuses

by

Steffen Sassalla
Matr.-Nr. 7111002

Thesis Supervisor:
Prof. Dr. Eric Bodden, Prof. Dr. Gregor Engels

Paderborn, April 19, 2021

Zusammenfassung. Bereits durchgeführte Studien haben gezeigt, dass nicht nur kryptografische, sondern auch nicht-kryptografische APIs falschverwendet werden. In dieser Arbeit evaluieren wir, ob das statische Analysewerkzeug CogniCrypt und die domänen-spezifische Sprache CrySL, beide einst entwickelt für die Erkennung von API-Missbrauch, auch auf nicht-kryptografische APIs anwendbar sind. Für die Evaluierung präsentieren wir ein Klassifizierungsrahmen für API-Verwendungsbeschränkungstypen, basierend auf bereits durchgeführten Studien. In Kombination mit 88 klassifizierten API-Falschverwendungen haben wir die Fähigkeiten von CrySL und CogniCrypt gemessen. Unsere Evaluation hat gezeigt, dass ein signifikanter Unterschied in Verwendungsbeschränkungen zwischen kryptografischen und nicht-kryptografischen APIs besteht. Da CrySL auf die Domäne der Kryptografie zugeschnitten ist, verfügt es im Allgemeinen über nicht genügend Ausdrucksstärke, um die korrekte Verwendung von nicht-kryptografischen APIs spezifizieren zu können. Wir schlagen daher kleinere Erweiterungen für CrySL vor, die es theoretisch erlauben, mithilfe von CogniCrypt die meisten von uns untersuchten Falschverwendungen zu erkennen.

Abstract. Previous studies have depicted that not only cryptographic but also non-cryptographic APIs are likely to be misused. In this work, we evaluate whether the static analysis tool CogniCrypt and the domain-specific language CrySL, once developed for detecting API misuses in code, are also applicable to non-cryptographic APIs. Therefore, we present a comprehensive classification framework of API usage constraint types based on several previously conducted studies. In combination with 88 classified API misuses, we measured the capabilities of CrySL and CogniCrypt. Our evaluation has shown that there is a significant difference in usage constraints of cryptographic and non-cryptographic APIs. Since CrySL is tailored to the domain of cryptography, it does not provide enough language constructs to specify the correct usages of non-cryptographic APIs in general. We propose some minor extensions to CrySL that theoretically allow CogniCrypt detecting most of the misuses we have investigated.

Keywords. API misuses, classification, API misuse detection, domain-specific language, static analysis

A professor and his secretary at the University of Paderborn, who used to be responsible for talking to lateral entrants, told me I'd better not take up the course because it wasn't feasible for me. Now I am about to graduate, almost in the standard period of study. I'm glad I chose this path anyway.

> *It is harder to crack prejudice than an atom.*
>
> Albert Einstein

Contents

1 Introduction

A desirable aim in the field of software development is to reuse prior implemented functionality. From a developer's point of view, it is easier to reuse functionality that has already been written; and, from a business point of view, it saves time and money. In software engineering, the developer can take advantage of previously written functionality by using application programming interfaces (APIs). Thus, it is common that a developer is offered a comprehensive bundle of APIs from the outset when beginning to use a programming language. In the case of the programming language Java, for example, the *Java Class Library (JCL)*[1] is included — a set of APIs that, for instance, provide basic functionality for the convenient use of data structures and allow comfortable communication with I/O devices. But it is not only vendors like Oracle that provide APIs; it is also the community of developers that supports the idea of sharing functionality (i.e., of providing APIs to each other). For example, the Maven repository[2] consists of thousands of APIs provided by a huge community that a developer can benefit from using. Thus, it is not surprising that, on average, a Java project depends on 14 different libraries [WWL+18], which is the consequence of excessive use of APIs. As a result, a developer evolves more and more into a composer of an arrangement of APIs instead of implementing the desired functionality from scratch.

However, the use of APIs has its drawbacks. Recent studies show that developers often face several difficulties [SHA15, NKMB16, LCWZ14, Isl20, GWL+19], such as the lack of appropriate documentation, the complexity of the API, or even an inadequate level of abstraction [NKMB16]. As a result, developers tend to incorrectly integrate APIs in their code which leads to misbehavior, errors, or even cause a program to crash. The potential consequences are even more alarming in the case of cryptographic APIs that are commonly used to protect sensitive data and prevent it from being eavesdropped. Faulty integration of cryptographic APIs creates a vulnerable attack surface, for example, to tap into sensitive data not intended to be read in plaintext by unauthorized parties. Lazar et al. [LCWZ14] found that developers themselves introduce 83% of all vulnerabilities through the incorrect use of cryptographic APIs. In recent times several approaches have been advanced to tackle this abuse. Most of these approaches do not focus on the prevention of incorrectly applied cryptographic APIs but on the detection of misuses [Ama18].

One of such detection approach is CogniCrypt [KNR+17], a static analysis tool that assists developers in the detection of incorrectly used cryptographic APIs. CogniCrypt can be

[1] https://docs.oracle.com/javase/8/docs/api/allclasses-frame.html
[2] https://mvnrepository.com/repos

conveniently used by performing the static analysis using an Eclipse[3] plugin. Compared to other analysis tools, CogniCrypt uses API specifications which are declared in a whitelisting approach. These hand-crafted API specifications are recorded in the domain-specific language CrySL. With the aid of CrySL, cryptography experts can provide specifications of their API in an easy-to-use and powerful language. For example, a cryptography expert can specify the methods that need to be called in a very specific order to guarantee a secure use of the API. The authors of CrySL already specified rulesets for the *Java Cryptography Architecture (JCA)*[4], *BouncyCastle*[5] and *Google Tink*[6]. With these rulesets already specified, the authors demonstrated the effectiveness of CrySL and CogniCrypt by measuring precision (i.e., the proportion of findings that constitute actual API misuses) and recall (i.e., the proportion of findings of all API misuses) of 50 randomly selected Android apps from *AndroZoo* [ABKT16]. The results show a high precision (92.6%) and recall (86.2%) in their manual assessment.

The proper use of cryptographic APIs is crucial for secure software development. However, recent studies [Ama18, ANN+16, Li20, LWY+18] have shown that the misuse of APIs goes even beyond cryptographic APIs. Amann et al. [ANN+16] examined the prevalence of API misuses in several bug datasets, i.e., datasets such as *BugClassify* [HJZ13] or *Defetcts4J* [JJE14]. They found that 89 of all 1.189 reviewed bugs were API misuses (7.49%), but 69.5% of these caused the program to crash. Although the number of API misuses found in the datasets was small, the high probability of causing a crash underlines the need for API misuse detectors. Furthermore, Li [Li20] performed a systematic and extensive empirical study of API misuses in the wild. They found that 50.6% of all bug-fixing commits published to GitHub[7] between 2011 and 2018 were related to API misuses. Even if high-quality documentation is provided, it may still not be enough to avoid API misuses altogether [ANN+19].

The tandem of CogniCrypt and CrySL have already proven themselves with their comparably high precision and recall for cryptographic APIs. At first glance, CrySL provides the necessary expressiveness to specify the correct usages of non-cryptographic APIs. Consider, for example, a correct application of the *java.io.FileReader*[8] shown in figure 1.1. For the moment, exception handling is not taken into consideration. Whenever the file is successfully opened, the file's content can be *read()* and following the usage, the resource needs to be released by invoking *close()*. In contrast, figure 1.2 shows a misuse where *close()* is not called in the end. CrySL allows the specification of the requirement of calling *close()* at the end of every usage of *FileReader*. Furthermore, by providing CogniCrypt the respective CrySL rule, the misuse can be correctly indicated in code.

This promising example now raises the question of whether CrySL and CogniCrypt have

```
1 FileReader r
2 = new FileReader("file");
3 r.read();
4 r.close();
```

Figure 1.1: A correct usage of *java.io.Filereader*. The resource is released at the end of usage.

```
1 FileReader r
2 = new FileReader("file");
3 r.read();
4 // missing call to close()
```

Figure 1.2: A misuse of *java.io.Filereader*. The resource is not released at the end of usage by calling *close()*.

[3] https://eclipse.org/
[4] https://docs.oracle.com/javase/8/docs/technotes/guides/security/crypto/CryptoSpec.html
[5] https://bouncycastle.org/java.html
[6] https://github.com/google/tink
[7] https://github.com/
[8] https://docs.oracle.com/javase/8/docs/api/java/io/FileReader.html

the capabilities to apply to non-cryptographic APIs in general.

1.1 Problem Statement

This thesis is guided by five research questions (RQs) to assess CrySL's and CogniCrypt's capabilities outside the domain of cryptography. The first two research questions are independent of CrySL and CogniCrypt but elaborating the necessary theory needed for an assessment.

RQ1: *How is the term API misuse defined, delimited, and made tangible?*
Before we can address the question of the applicability of CrySL and CogniCrypt to non-cryptographic APIs, it is necessary to define the term API misuse and its scope. This in turn requires a clear definition of what an API actually is, and an exploration of the root cause of API misuses — i.e., to identify the nature of API misuses — as well as the precise definition of an API misuse. Only by having a clear view of API misuse's overall scope, we can evaluate CrySL and CogniCrypt with the concrete scope in mind.

RQ2: *What classification framework of API misuse types can be used to evaluate the capabilities of a specification language like CrySL and an analysis tool like CogniCrypt?*
Having precisely defined the terms API and API misuse (RQ1), there is now the question of what types of API misuses can actually occur. Therefore, it is necessary to first conduct a literature survey of existing classification frameworks in order to either use an existing classification framework or devise a classification framework from several resources.

RQ3: *What are the capabilities of CrySL and CogniCrypt?*
With the classification framework elaborated in RQ2, we can systematically evaluate the capabilities of CrySL and CogniCrypt. This includes identifying and using appropriate datasets that contain API misuses, reviewing and reclassifying the API misuses based on the elaborated classification framework, and finally applying them systematically to CrySL and CogniCrypt, to finally evaluating the outcomes.

RQ4: *What are the precision and recall of CogniCrypt when applied to non-cryptographic APIs?*
If RQ3 reveals that CrySL and CogniCrypt are applicable to non-cryptographic APIs in most cases, the next step is to quantify their capabilities by evaluating precision and recall.

RQ5: *If CrySL and CogniCrypt fail to identify certain API misuses, how can they be improved upon?*
If RQ3 reveals weaknesses of CrySL or CogniCrypt related to certain API misuse types from the classification framework, the question arises what targeted improvements can be carried out to ensure the applicability to non-cryptographic APIs. It would be outside the scope of this thesis to include a detailed study into the implementation or feasibility of any suggested improvements. Nevertheless, we will underline the necessity of the suggested improvements, and provide a theoretical argument for their implementation.

The scope of this thesis is limited to the programming language Java and its *Java Platform SE 8*[9], since CogniCrypt was developed based on this Java version. Although the concepts and definitions introduced throughout this thesis may be transferable to other programming languages and newer Java versions as well, they are introduced with their concrete application to Java SE8 in mind.

1.2 Structure of this Thesis

In chapter 2, we first provide an overview of the different definitions surrounding the term API misuse, as well as the different classification frameworks that have been put forth to categorize the various types of API misuses. We will also argue why it is not sufficient to rely on previous work, their definitions and classification frameworks, but to introduce our own. In chapter 3, we introduce to the domain-specific language CrySL and the static analysis tool CogniCrypt. This also includes a clarification of CogniCrypt's and CrySL's classification in the world of analysis tools. In chapter 4, we present the evaluation results based on our elaborated classification framework, as well as the concrete identification of the weaknesses and strengths of CrySL and CogniCrypt. We highlight the results by comparing the domain of cryptographic APIs with that of non-domain-specific APIs. In chapter 5, we introduce theoretical arguments for improving both CrySL and CogniCrypt to mitigate the discovered weaknesses based on the evaluation. We discovered relevant open research questions beyond the scope of this thesis; hence we present them in chapter 6. We conclude this thesis in chapter 7 by revisiting and summarizing the corresponding answers for the research questions posed.

[9] https://docs.oracle.com/javase/8/docs/api/

2 Violations of API Usage Constraints

In modern software development, the reuse of functionality is a desired approach because the developer does not have to implement it entirely from scratch. In object-oriented languages such as Java, the functionality is usually recorded in multiple classes bundled and offered to the developer as a library. Classically, a library can be divided into two areas: the public interface and the private implementation [METM12]. The private implementation is the part that actually implements the functionality of the library. However, concrete implementation details are usually not exposed to the developer (i.e., to the library's user). Throughout this thesis, we will not further consider the part of private implementation but the public interface, as we focus on its misuses. The public interface exposes software elements (e.g., classes and methods) to the outside world, making the implemented functionality accessible. In the literature, these public interfaces are known as **application programming interfaces (APIs)**. Furthermore, we refer to an **API class** as a class from the API and an **API object** as its instantiation.

At first glance, the widespread use of an API (and its instantiated objects) seems simple because in predominant cases, only method calls are necessary to get the desired functionality [ZM19]. However, it can be much more complicated. For instance, developers must consider certain restrictions in environments (e.g., multi-threading) or be aware of specific properties (e.g., when performance plays an important role). To support developers, APIs are delivered with documentation, which can be very diverse in nature [METM12]. For example, parts of the documentation can be related to typical use case scenarios, code snippets, constraints, or even performance discussions.

We now have a clear understanding of the definition of an API. However, we do not know how it relates to API misuse, nor do we know the root cause of API misuse, and what concrete API misuse types exist. Therefore, in section 2.1, we provide an overview of previous works that have declared definitions related to API misuses, elaborated on the root causes, and focused on classifying the different misuse types. As a result, in section 2.2, we will explain why relying on previous studies is not enough to answer the question for the tangibility of the term API misuse (**RQ1**) and the question for a classification framework of the different API misuse types (**RQ2**). Therefore, in section 2.3, we introduce our own definitions but do so with respect to previous research to answer **RQ1**. With a clear view of the definitions around the term API misuse, we will then contribute a comprehensive classification framework in section 2.4 to answer **RQ2**. In section 2.5, we discuss the differences between our classification framework and others. Since our classification framework is a collection of contributions that also present empirical studies of API misuse, we provide in section 2.6 a direction for the distribution of each type of our classification framework.

2.1 Related Work - A Survey of Definitions and Perspectives

Whenever an API is used, developers need comprehensive and explanatory documentation to use it properly. Thus, research in the past focused on elaborating on the different kinds of API documentation. This section provides an overview of several studies focused on different aspects of API documentation and even beyond. We consider previous work from two perspectives. From a first perspective, we classify previous work by its range on different (implicit) types of API documentation and its noncompliance. For example, Robillard and Maalej [MR13] focused on every part of an API documentation, whereas Lv et al. [LLY+20] only considered those parts of documentation for which noncompliance would result in errors or misbehavior. From a second perspective, we show that almost every study provides its own definitions (denoted in bold letters).

2.1.1 API Documentation Types and Definitions

Dekel and Herbsleb [DH09] researched how developers could better be made aware of *usage directives*. They defined **usage directives** as parts of API documentation that capture non-trivial, infrequent, and possibly unexpected information. They introduced ten different types of usage directives. For example, one type is called *Restrictions*, which addresses the context in which the API method should or should not be called. For instance, an API method should only be called in the context of debugging purposes. To highlight *usage directives* directly in the development environment, they developed an Eclipse[1] plugin called *eMoose*.

Bruch et al. [BMM10] investigated parts of API documentation from the perspective of extensibility since API classes can also be inherited and can thus be specialized or re-implemented further. They defined parts of API documentation related to extensibility and inheritance as **subclassing directives**. For example, the subclassing directive *subclasses may extend this method* that requires subclasses to call the super method. In contrast, the directive *subclasses may re-implement this method* aims at re-implementation, and thus, subclasses must not call the super method. In total, they introduced four types of such subclassing directives.

Based on the studies from Dekel and Herbsleb and Bruch et al., Monperrus et al. [METM12] derived a classification framework of API directive types. They defined **API directives** as natural-language statements of API documentation that describe how to use an API correctly and optimally. Their classification framework comprises 26 different API directive types. Moreover, they performed an empirical study on the variety of API directive types in the wild, based on the documentation of three Java libraries, namely the *Java Class Library*[2], *JFace*[3] and the *Apache Commons Collections*[4]. With the analysis of 4.561 API elements (i.e., documentation of interfaces, packages, classes, methods, and fields) in total, they showed the prevalence of each type, respectively.

Robillard and Maalej [MR13] empirically elaborated a comprehensive taxonomy of so-called API knowledge types. An **API knowledge type** is a particular pattern of knowledge that classifies a specific part of API documentation. For example, there is the API knowledge type called *Directive*, which specifies what developers are allowed or not allowed to do with the API. Another example is the *Quality* type that describes non-functional requirements of the API (e.g., performance implications). In total, they introduced 12 of these types. However, the study of Robillard and Maalej investigated the various parts of API documentation in a broader sense than Monperrus et al. did.

[1] https://eclipse.org/
[2] https://docs.oracle.com/javase/8/docs/api/allclasses-frame.html
[3] https://wiki.eclipse.org/JFace
[4] https://commons.apache.org/proper/commons-collections/

2.1.2 API Usage Constraint Types and Definitions

In addition to studies that focused on the various types of API documentation as a whole, there are studies targeting only a subset of them. This subset refers only to parts of API documentation that a developer must adhere to in order to avoid encountering misbehavior, errors, or vulnerabilities.

Li et al. [LLS+18] narrowed down the set of API documentation parts to so-called API caveats. An **API caveat** is a directive which noncompliance would likely incur unexpected program behaviors or errors. According to Li et al., API caveats are scattered across multiple API documentations in sometimes lengthy textual descriptions. Therefore, they proposed a natural-language processing technique to link API caveats sentences in an *API caveats knowledge graph*. Moreover, they introduced a classification framework of syntactic patterns that comprises ten types of API caveats.

Just like Li et al., Lv et al. [LLY+20] considered only the parts of API documentation for which noncompliance would result in severe consequences such as authentication errors or NULL-dereference. They call these types of API documentation **integration assumptions (IAs)**. Every IA has its constraint related to pre-conditions (e.g., limit on parameter lengths), post-conditions (e.g., a contract on the return value of a method call), or to the invocation context (e.g., a method must be called from a specific thread). They proposed a technique to automated assumption discovery and verification derivation from library documents, called *Advice*. Unfortunately, they did not provide any fine-granular classification framework.

Similar to Lv et al., Nguyen et al. [NVN19] classified constraint types that lead to serious programming errors but without having a concrete definition for them. They categorized such constraints as temporal order (e.g., API method calls are expected to be called in a particular order), pre-conditions and post-conditions (similar to Lv et al.), argument value (e.g., only a specific type is allowed to be passed), and finally, exception (e.g., certain exceptions need to be handled). Compared to Lv et al., they also considered the proper exception handling and the order of method calls in which they are intended to be called. They proposed a novel approach called *Statistical Approach for API Misuses* (SAM) that tries to detect noncompliance of the mentioned constraints in code.

In contrast to Lv et al. and Nguyen et al., Saied et al. [SSD15] investigated so-called **API usage constraints**. According to Saied et al., API usage constraints are — and should be — captured in the respective API documentation. They introduced four types, namely *Nullness not allowed* (i.e, passing null value to a method call results in failures), *Nullness allowed* (i.e., passing null value has a certain semantics), *Range limitation* (i.e., a passed numeric value is restricted to a set of allowed values) and *Type restriction* (i.e., the parameter type is restricted). Interestingly, all introduced classifications except *Nullness allowed* restricts the API usage, and noncompliance would result in runtime failures. Saied et al. showed that such API usage constraints are frequently in code but not always documented.

The problem of the non-existence of important API usage constraints in API documentation was also addressed by Amann [Ama18] in their Ph.D. thesis. They defined **API usage constraints** to be (implicit) constraints not enforced by the compiler, such as correct typing, but constraints enforced by the API, for which noncompliance would result in runtime errors. Thus, an *API usage constraint* does not necessarily need to be captured by the API documentation. One part of their work summarizes Monperrus et al.'s comprehensive classification framework of API directives to provide an overview of categories that are API usage constraints.

In contrast to the definition of Amann, Ren et al. [RSX+20] defined **API usage directives** as "contracts, constraints, and guidelines that specify what developers are allowed/not allowed to do with the API". As a result, Ren et al.'s definition also comprises parts of the API documentation in which noncompliance does not necessarily result in errors or misbehavior.

In addition, several (mostly empirical) studies [SHA15, CPNM18, BBA09, BKA11] investigate only very certain constraints, mostly referred to as **object protocols**. According to Beckman et al., *object protocols* are "protocols dictating the ordering of method calls on object of a particular class". A prime example is the *java.util.FileReader*[5] API class (cf. figure 1.1 and 1.2) where *read()* is only allowed to be called if the resource was opened successfully. At the end of the *FileReader*'s usage, the resource needs to be released by invoking *close()*. After *close()* has been called, no further methods are allowed to be called on the API object because it is considered to be "deactivated". Such a predefined method call order is defined as an *object protocol*.

2.1.3 API Misuse Types and Definitions

So far, we have only considered studies investigating the spectrum of API directives and API usage constraints. As shown in figure 2.1, API usage constraints are a subset of API directives for which violation results in misbehavior, errors, or vulnerabilities at runtime (cf. section 2.1.2). The dashed line divides API directives as well as API usage constraints into two areas: they can either be implicit or documented. Since the designer of an API is not given language constructs provided by the programming language to force the API user to comply with API usage constraints [Ama18], they can be (unintentionally) violated. Such a violation is mostly referred to as **API misuse** in the literature. We will introduce precise definitions in section 2.3. The classification, detection, and mitigation of API misuses have been studied in various specific fields of research — for instance, cryptography [NKMB16, LCWZ14, Ste20, EBFK13], machine learning [Isl20], stream APIs [KTBR20], and even biometric APIs [JCX19], as well as in the overall analysis of API misuses [Ama18, Li20, SBLV+19, KS14, KDP+19, HP04]. Furthermore, API misuses are not tied to just one specific programming language like Java. There are also studies focusing on API misuses in different languages like C [GWL+19] or Python [Isl20]. It is thus a more general problem using APIs. However, we observed in the literature different kinds of definitions for the term API misuse.

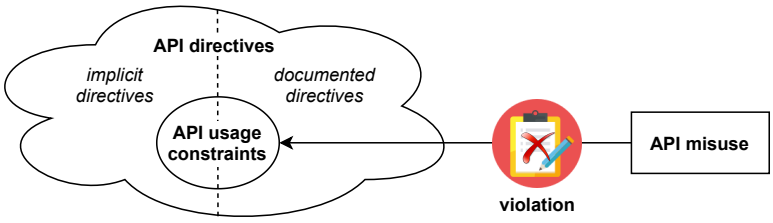

Figure 2.1: Overview of relations between the different termini we observed in the literature.

Luo et al. [LWY+18] investigated the continuous and rapid development of the *Android Platform APIs*[6], which are often used in the development of Android applications. They find that Android applications (apps) often cannot keep up with the rapid development of the underlying *Android Platform APIs*. As a result, the developed apps can run into compatibility, security, or reliability problems. They showed that APIs are not necessarily persistent and thus, the underlying documentation, including API usage constraints, are also continuously expanded and changed. They define an **API misuse** as the violation of an API's contract, resulting from an update to the underlying API, but improperly maintained API calls.

Khatchadourian et al. [KTBR20] were interested in how Stream APIs are misused and why

[5] https://docs.oracle.com/javase/8/docs/api/java/io/FileReader.html
[6] https://developer.android.com/reference

developers are struggling with their correct use. They define bugs related to Stream APIs as **Stream Misuses**, where the term "bug" is no further specified. From 22 projects using streams, they identified different classifications of Stream Misuses by analyzing Git[7] commits. They derived a classification framework comprising 15 different categories. Interestingly, one category is called *Perf* (poor performance), which aims at Stream Misuses resulting from improper application of the Stream API without resulting in any error but performance implications.

Especially in the domain of cryptography, researchers are interested in API misuses [NKMB16, Ste20, EBFK13, LCWZ14, CNKX16]. Nadi et al. [NKMB16] found that cryptographic APIs' low level of abstraction leads to incorrect usage. As a result, the API becomes too complex to use to achieve a specific goal. For example, there is the need to compose several method calls over different API objects to encrypt a file correctly and securely. Egele et al. [EBFK13] conducted an empirical study in the field of cryptography with a focus on Android apps. They found that 88% of all apps have at least one mistake using cryptographic APIs. Unfortunately, they did not provide any precise definition of an API misuse related to cryptography. However, they showed that an API misuse does not necessarily result in errors or crashes but sensitive vulnerabilities causing possible security breaches.

According to a study of Amann [Ama18] conducted in 2018, they were the first who systematically defined the problem space of API misuses and the first who empirically investigated the prevalence of API misuses in bug datasets. They introduced a comprehensive classification framework of 16 different API misuse categories, so-called API-Misuse Classification (*MuC* [ANN+19]). They defined an **API misuse** as a violation of an (implicit) API usage constraint (cf. section 2.1.2). Although they summarized the findings of Monperrus et al. [METM12] regarding API usage constraints, they did not directly derive API misuse categories from Monperrus et al.'s work. They rather built a classification framework based on the findings of Monperrus et al. and the results from experiences of API misuses they studied. To gain overall knowledge of the nature of API misuses, they investigated over 1.200 bug reports from bug datasets like *BugClassify* [HJZ13] or *Defect4J* [JJE14]. As a result, they derived *MuC* from a total of 164 identified API misuses. In addition, they provided their own dataset with pre-classified API misuses based on *MuC*, the so-called *MUBench Dataset*[8]. Together with *MUBench* — a benchmarking tool to assess the capabilities of detectors that indicate API misuses — they provide an overall powerful toolbox in the field of API misuses and detectors.

Since Amann only conducted a study based on API misuses from bug-datasets, *MUBench* might therefore be limited for evaluating detection techniques on API misuses. Thus, Li [Li20] conducted the first large-scale empirical study of API misuses in 2020. They defined an **API misuse** as "[...] code edit operations related to some APIs in a bug fix". Although their definition is in stark contrast to Amann's, they followed up on *MuC*, refined the classification framework by restructuring and introducing new categories. Moreover, they mined all bug-fixing commits of Java projects from GitHub[9] in the timeline between 2011 and 2018. Subsequently, Li screened the bug-fixing commits by their commit message containing keywords like "fix", "bug" or "issue". Although this approach might also include commits that are not necessarily related to real bug fixes, they randomly selected a sample of 100 commits. As a result, 94% of the analyzed commits of the sample were real bug-fixing commits. According to Li, this confirms that the mined bug-fixing commits target real bug-fixes and therefore argued their approach is adequate. From over three million investigated changes (3.197.593) in the respective commits, Li found 50.6% were involved in API misuses demonstrating that developers heavily struggle with API misuses in modern software development.

[7] https://git-scm.com/
[8] https://github.com/stg-tud/MUBench/tree/master/data
[9] https://github.com

2.2 Motivation

As shown in the previous section, several studies provide fundamental theory. They already present comprehensive classification frameworks of the different types of usage constraints and misuses [Ama18, METM12, Li20, BKA11]. However, we observed that the definitions surrounding the term API misuse are scattered across several studies in different versions, sometimes targeting different aspects of an API misuse. In fact, researchers classify API misuse types by (i) the overall definition of an API misuse in the specific domain and (ii) by the need of their work. This section provides two reasons why we study the term API misuse and its root causes again.

1. Scattered across several studies, different approaches have already been introduced to classify usage constraints. However, some work focuses on specific areas, such as object protocols [SHA15, CPNM18, BBA09, BKA11], while others try to classify — sometimes roughly — the spectrum of API directives, usage constraints, or API misuses [METM12, Ama18, Li20, NVN19, SSD15, DH09, SHA15, ZM19, BBA09, LWY+18, KTBR20, LLY+20, LLS+18, MR13, SBLV+19, KS14, KDP+19, HP04, Rat13] (cf. section 2.1). The most comprehensive summary of relevant misuse types is yet provided by Amann [Ama18]. However, there is information missing due to the abstraction level they chose. For instance, Monperrus et al. [METM12] mentioned the *Method Parameter Type Directive* which mandates the allowed type of method parameters. For example, the *compareTo(Date d)* method of *java.sql.Timestamp*[10] states: *"Compares this Timestamp object to the given Date, which must be a Timestamp object."*. Although Amann mentioned this type in their summary of API directives, it gets lost in their API misuses' derivation without further mentioning it. We want to be sure not to miss certain root causes of API misuses. Thus, our approach is first to define all necessary terms surrounding API misuses (**RQ1**) to have a clear view of its meaning. This allows us to collect previous contributions, to provide one comprehensive framework (**RQ2**). With this framework, we can precisely assess the capabilities of CrySL and CogniCrypt based on state-of-the-art knowledge (**RQ3**).

2. We show that usage constraints can be associated with specific parts of a method call. For instance, usage constraints of type *Post-Call Directive* [METM12] categorizes requirements for method calls that must be invoked on the returned object. Thus, this usage constraint type only targets the return value of a method call. We can also observe usage constraint types associated with the invoked method itself, the return value, the passed arguments, and the context in which the API is used. Such localization helps to identify the scope of individual API misuses to help in the evaluation part of CrySL and CogniCrypt (**RQ3**) and to consider improvements that might be made to the language design of CrySL or analysis techniques on CogniCrypt (**RQ5**).

2.3 Definitions

We can basically divide APIs into two areas in terms of their application domain. Firstly, there are APIs with a focus on a particular domain. For example, cryptographic APIs implement cryptographic algorithms and provide basic functionality for data security. On the other hand, some APIs are domain-independent. Consider, for instance, the prime example *java.lang.String*[11]. It provides overall functionality to work with strings that are used domain-independently. This view on APIs leads to the following definition:

[10] https://docs.oracle.com/javase/6/docs/api/java/sql/Timestamp.html
[11] https://docs.oracle.com/javase/8/docs/api/java/lang/String.html

Definition 2.1. *A **domain-specific API** offers functionality tailored to a specific domain. Its domain determines the achievable goal and application rules of a domain-specific API. In contrast, **non-specific APIs** are not tailored to a domain, nor do they determine a specific goal associated with their use.*

Developers need a deep understanding of how an API is intended to be used. Therefore, the API documentation is a core requirement that provides code examples, conceptual overviews, definitions of terms, programming guidelines, known bugs, and documented workarounds [Kra99]. In fact, API documentation is manifold in its nature [MR13, METM12]. Essential for developers are contracts enforced by the API. Such contracts, also called *API directives*, describing what a developer is or is not allowed to do [MR13]. Monperrus et al. [METM12] described them as "[...] natural-language statements that make developers aware of constraints and guidelines related to the usage of an API". They consider API directives to be captured in the respective API documentation. However, the documentation does not necessarily need to be recorded in a high quality, and thus it may be incomplete or contradictory [Rat13, NVN19, SSD15]. In the worst case, the complete documentation is missing. Furthermore, sometimes the API takes domain knowledge for granted [NKMB16], which may not be further mentioned in the documentation. Therefore, we extend the definition of API directives not only to the underlying API documentation but also to *implicit* statements. This conceptual expansion to Monperrus et al.'s [METM12] view on API directives leads us to the following definition:

Definition 2.2. *An **API directive** is a natural-language statement related to guidelines or constraints that describes how to use an API correctly and optimally. It can be part of the underlying documentation of an API. However, an API directive can also be implicit, for example, because of incomplete documentation or domain-specific knowledge, which the API takes for granted.*

Because API directives can also be guidelines (e.g., hints for performance improvements), we can further divide API directives. For a developer, constraints especially are important to consider. A constraint is a contract that narrows down the actual use of an API. For example, an API may require a method to be called only under certain conditions. In the case of *java.util.Iterator*[12] it is required to call *next()* only if the iterator contains at least one other element. An API designer cannot force a developer to adhere to these constraints because the programming language does not provide any language constructs for its compliance (e.g., correct typing enforced by the compiler [Ama18]). In conclusion, this implies the following definition of an API usage constraint based on the definition of Amann [Ama18]:

Definition 2.3. *An **API usage constraint** is an API directive that restricts the actual use of an API. These restrictions are not enforced by the programming language itself, such as correct typing. Because API usage constraints are API directives, they are imposed by the API designer/expert.*

Note that an API usage constraint is tailored to the perspective of an API designer/expert. Here, an API designer/expert is responsible for the API design, which includes imposing the constraints on it. This perspective is important as API usage constraints are API directives and therefore not imposed by any API user. For example, it is not considered an API usage constraint if the user of *java.lang.String*[13] imposes the restriction of using only UTF-8 charset because the development context requires such for any reason.

[12] https://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html
[13] https://docs.oracle.com/javase/8/docs/api/java/lang/String.html

The developer can violate API usage constraints, for example, because of a lack of domain knowledge [NKMB16] or improper documentation [Rat13]. A violation of an API usage constraint harms the program's further course as it leads to misbehavior of the API, errors, crashes, or even worse, security vulnerabilities. Therefore, we extend the definition of Amann [Ama18] to the following:

Definition 2.4. *An **API misuse** is the violation of an API usage constraint. Such violation leads to misbehavior of the API, errors, crashes, or vulnerabilities.*

An example of an API misuse is to call *next()* without ensuring the *java.util.Iterator*[14] contains at least one element, which can be checked by calling *hasNext()*. Another example is to not release a recently opened resource by calling *close()* after the end of using *java.io.FileReader*[15] (cf. figure 1.2).

In conclusion, the definitions presented in this section have answered the first research question (**RQ1**) by discussing the relevant aspects surrounding API misuse and deriving definitions concerning previously conducted studies. The provided definitions narrow down the term API misuse, as it is a violation of an API usage constraint. An API usage constraint is imposed by an API designer/expert and does not necessarily need to be captured in the respective documentation. Thus, API usage constraints may also be in domain-specific knowledge that the API takes for granted.

2.4 Taxonomy of API Usage Constraints

To accurately assess the capabilities of a specification language like CrySL and a static analysis tool like CogniCrypt, we need to elaborate on the different types of API usage constraints (**RQ2**), which we will cover in this section. Please recall that we consider an API usage constraint to be a concrete constraint imposed by the API. Accordingly, an **API usage constraint type** is a set of API usage constraints that share the same kind of constraints. For example, we represent a concrete API usage constraint such as "the passed argument must not be null" by an API usage constraint type (or short constraint type) related to null values and method parameters, which is called *Pre-Null-Check*.

In contrast to other studies [Ama18, Li20], we do not provide a classification framework of API misuse types but of API usage constraint types. This is because API usage constraints are the root cause of API misuses and only their violation results in API misuses (cf. definition 2.4). Without having API usage constraints on APIs, there would be no API misuses because the API is not restricted in its actual use. Moreover, we localize each API usage constraint type by the parts of an API method call (i.e., the API usage).

2.4.1 Constraint Types by Parts of an API Method Call

Since Monperrus et al. [METM12] introduced the most comprehensive and fine-granular collection of API directive kinds and an empirical study on their prevalence in API documentation, we take his contribution as a basis. Therefore, we will consider all their mentioned API directive kinds that actually are API usage constraint types. We then enrich Monperrus et al.'s findings by contributions of previous other studies [Ama18, Li20, BKA11, NVN19, Rat13, CPNM18, SSD15, ZM19, DH09, SHA15, BBA09, LWY+18, KTBR20]. As a result, we introduce six totally new or more fine-granular types additional to the API usage constraint types already elaborated by Monperrus et al., i.e., *High-Level Constraints*, *Post-Null-Check*, *Controlling Method Call*,

[14] https://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html
[15] https://docs.oracle.com/javase/8/docs/api/java/io/FileReader.html

Threading, *Argument State*, and finally, *Pre-Null-Check* (annotated with an asterisk in figure 2.2).

Since interaction with APIs is due to method invocations [ZM19], we will look at API usage constraints from a method call perspective. Therefore, we will assign each API usage constraint type to the different parts of an API method call (cf. figure 2.2). We furthermore consider an API method call to be a method invocation either on the API class or on the respective instantiated object (i.e., the API object). This perspective on API usages allows us to clearly separate and localize API usage constraints based on the use of APIs. We consider an API method call to have three relevant parts: the receiver, the method call itself, and the passed arguments. In addition, there are also API usage constraint types associated with multiple parts of the API method call (dashed boxes without colors). A detailed mapping of the API usage constraint types elaborated by Monperrus et al. to our types can be seen in table 8.1 in the appendix.

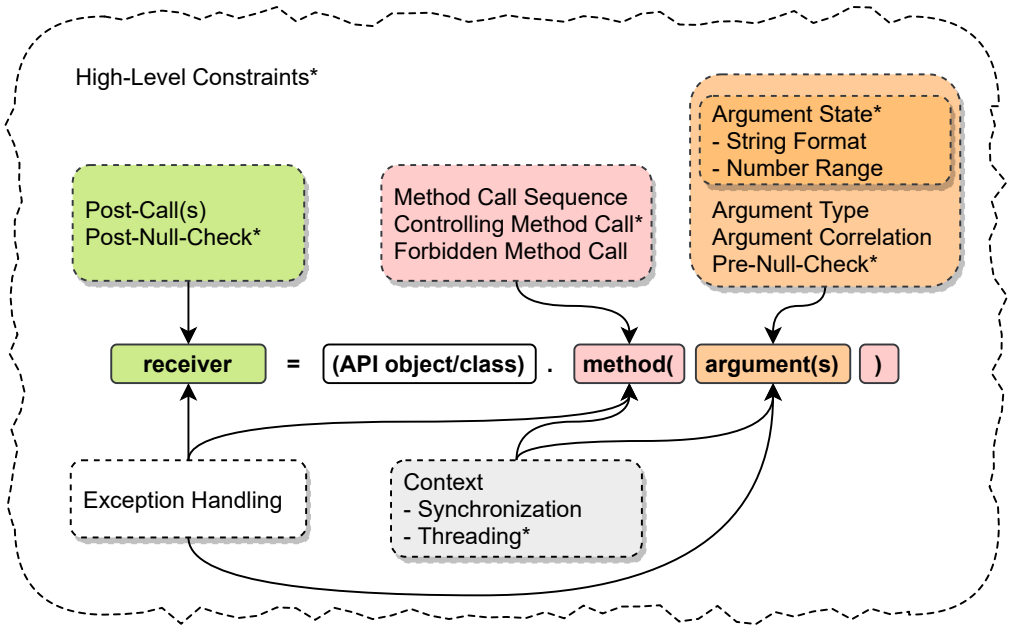

Figure 2.2: The figure shows the big picture of API usage constraint types associated with parts of an API method call. Types marked with an asterisk are additionally added to the work of Monperrus et al. [METM12]. Dashed colored boxes are specific to one single API method call part. Dashed boxes without being colored are API usage constraint types that span multiple parts of an API method call.

2.4.1.1 Receiver

We define the return value of an API method call as a receiver. An API can impose constraints even beyond the method call itself (e.g., requirements in the form of methods that need to be called on the return value). We can observe two types of API usage constraints associated with the receiver (cf. green boxes in figure 2.2).

Post-Call(s) constraints mandate calling methods on the receiver [METM12, NVN19, LLY+20]. Sometimes it is not possible to wrap the functionality in its whole in a single method. This can be the case when the desired functionality depends on the client application. For instance, the

static method *AlgorithmParameters.getInstance(algorithm)*[16] requires initializing the receiver via the call of *init()* [METM12]. Note that this API usage constraint type is a specialization of *Method Call Sequence*, but with the fact that the required calls do not necessarily refer to the same API class. Rather, the receiver may be typed differently. In total, 0.9% of all analyzed API elements contain such usage constraint [METM12].

Post-Null-Check comprises rather implicit usage constraints that require a null check on a receiver not to run into runtime errors [Ama18, Li20, NVN19]. We can observe such constraints whenever a method returns either a specified value or *null*, which is mostly returned when there is actually no specific value that can be returned. For instance, the API class *java.io.InputStreamReader*[17] exposes the method *getEncoding()* which returns the name of the character encoding being used by the stream or *null* if the stream is closed. However, since the return value is a string, the developer may continue to work on this return value; there is a need to check for *null* beforehand. This usage constraint type is not mentioned by Monperrus et al. [METM12]. Note that this kind of usage constraint is clearly distinguishable to *Post Calls(s)* because no method call is used to check for *null*.

2.4.1.2 Method Call

This section introduces all usage constraint types associated with the method call itself (cf. red boxes in figure 2.2). Sometimes, the API restricts the interaction with itself, or methods need to be called in a specific sequence to achieve the desired result. There are also cases where methods are not supposed to be called, although they are public.

However, we can observe mainly two reasons for constraints on the invocation of a method. Firstly, the context in which an API is used requires or restricts specific usage. This is discussed in section 2.4.1.4. Secondly, the state of the API object dedicates the allowed, required, or forbidden usage of itself [METM12], which we will discuss in the following.

Method Call Sequence are usage constraints enforcing the requirement to call methods in a particular sequence [METM12, CPNM18]. A method call sequence can be considered as a usage protocol [BKA11, DH09, SHA15, Rat13] (i.e., the usage constraint specifies a protocol that defines the correct usage in a particular state of the API object). Thus, the correct usage of method calls can be modeled with a finite-state-machine [BBA09, Rat13, ZM19]. In addition to the notion of a usage protocol, the term typestate is also often used in the literature, meaning that the typestate defines the permitted operations on the API object based on its state [Ste20].

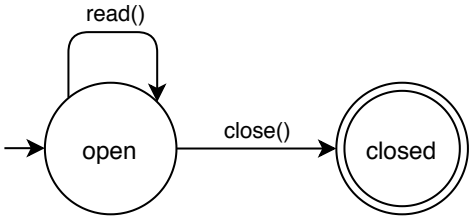

Figure 2.3: Simplified finite-state-machine of *java.utils.FileReader*[18]. Edges represent method calls. Circles denote states. The double bordered circle represents a final state so that the API's declaring type can be considered to be used correctly without further interaction.

[16] https://docs.oracle.com/javase/8/docs/api/java/security/AlgorithmParameters.html#getInstance-java.lang.String-

[17] https://docs.oracle.com/javase/8/docs/api/java/io/InputStreamReader.html

Consider for example the *java.io.FileReader*[18] and the simplified finite-state-machine displayed in figure 2.3. The *FileReader* is used to read character files. The state of its object is *open* when the file could be opened successfully. Only then *read()* of the character files is possible, which is represented as a looped edge to *open*. This can be repeated as long as the file is opened. At the end of the usage, there is a need to *close()* the resource so that there is a transition to the (final) state *closed*. Any derivation of calls specified in the finite-state-machine results in the (implicit) error state. This implies that the use of the *FileReader* is only correct if its object is in one of the defined final states at the end of the usage. Furthermore, the transitions to such a final state form a sequence. We can observe this type of usage protocol extensively in the context of cryptography, where a particular sequence of methods must be followed even across different APIs [NKMB16] — for example, to encrypt a file securely. In total, up to 12.2% of all analyzed API elements contain such usage constraint [METM12].

Besides the findings of Monperrus et al., Beckman et al. [BKA11] investigated usage constraints from the perspective of the typestate and the respective restrictions on method calls. They conducted a study on open-source Java projects and the *Java Class Library*[19] in which 7.2% of the investigated API classes define usage constraints related to method call sequence. Note that this study is not based on APIs' documentation — as Monperrus et al. did —; they conducted a static analysis on source code to detect usage constraints. Interestingly, 28.1% of all identified usage constraints by Beckman are related to the initialization of the API object: there is a need to call specific methods to initialize the API object for further usage. 25.8% are related to an API object's deactivation (i.e., a particular method call to transition the API object to a state where its use is no longer valid).

Controlling Method Call classifies constraints on a method call that is only allowed to be called if a condition is met beforehand [Ama18, BKA11, NVN19, BBA09, KTBR20]. Unlike *Method Call Sequence*, this usage constraint type aims to use method calls in conditional statements like an "if"-statement. Therefore, this type of method call controls — and safeguards — the further program flow depending on its return value. The return value can give information about the state of the API object and information if a method, including its arguments constellation, is allowed to be called. For instance, consider the *java.util.Iterator*[20] and the excerpt of its finite-state-machine in figure 2.4. In order to retrieve the *next()* element, there is a need to check whether the *Iterator* contains at least one element — to check the respective state of the

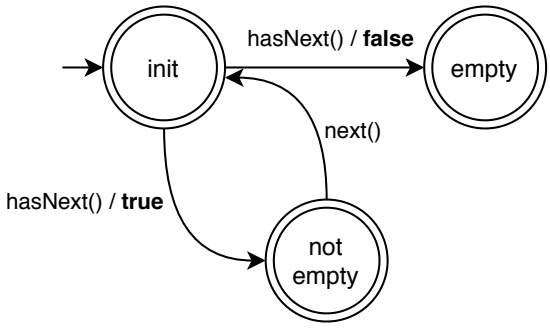

Figure 2.4: Simplified finite-state-machine of *java.util.Iterator*[20]. Edges represent method calls to transition to another state optionally with return values. The return value is decisive for the state the API object is in. Double bordered circles represent a final state.

[18] https://docs.oracle.com/javase/8/docs/api/java/io/FileReader.html
[19] https://docs.oracle.com/javase/8/docs/api/allclasses-frame.html
[20] https://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html

API object that allows the call of *next()*. This can be achieved by calling *hasNext()*. Only if *hasNext()* returns *true* we are allowed to use *next()*. Such controlling method calls can be seen as an interface for the developer to react to a specific state. As shown in figure 2.4, the return value of *hasNext()* reveals the state of its API object so that the developer can react accordingly. In fact, this is an extension of *Method Call Sequence*, as it can be seen as a sequence of method calls, but with the exception that further operations on the API object are dependent on the return value. Note that this usage constraint is not explicitly mentioned by Monperrus et al. [METM12].

Forbidden Method Call constraints specify that certain methods are not allowed to be called [METM12, Ama18]. In fact, this depends on the underlying state of the API object and is therefore a special case of *Method Call Sequence*. For instance, whenever an *Iterator* is generated from an *java.util.ArrayList*[21], concurrent operations like *add()* or *remove()* on the array list are forbidden. Besides, there are also special cases where a method may not be called in an API object's whole lifetime. For example, *java.util.logging.Logger*[22] prohibits calling *setParent(parent)* from the application code even though the method has been made public [METM12]. This can also be the case if methods are marked as deprecated but are still callable because of backward compatibility reasons [LWY+18]. Note that this usage constraint type is not explicitly mentioned by Monperrus et al. [METM12]. The most similar API directive type mentioned by Monperrus et al. is *Method Call Visibility Directive*, which addresses constraints on the calling context's method call. Thus, this directive overlaps with *Context* and *Threading* constraint types we introduce in section 2.4.1.4. Therefore, we can address the prevalence of *Forbidden Method Call* constraints with roughly 4.2% [METM12], but rather lower because of the overlapping.

2.4.1.3 Passed Arguments

This section introduces usage constraint types related to passed arguments of an API method call (brown colored box in figure 2.2).

Argument State constraints require the passed argument to be in a specific state or to have been correctly generated through a predefined protocol. Such predefined protocols specifying the composition of several API objects are also called *multi-object protocols* [Rat13, SHA15]. We can typically observe this type of usage constraint in the context of cryptography where several API objects need to be correctly composed [Ste20]. Consider for instance figure 2.5. In order to securely generate a message authentication code (MAC) by using *javax.crypto.MAC*[23], a securely generated key need to be passed to *init(key)* (lower finite-state-machine). This can be achieved by using *javax.crypto.KeyGenerator*[24] in no other way than the protocol (i.e., the method call sequence) specifies. This sequence is shown in the upper finite-state-machine of figure 2.5. Only if the *KeyGenerator* is not applied in any other way than the predefined protocol specifies, the MAC can be generated securely. Apart from multi-object protocols, we can observe special cases not perfectly fitting into this definition. This is, for example, if constraints are enforced on primitive types and string literals. We therefore further distinguish between two special cases (i.e., *String Format* and *Number Range*). However, because we tightly refer to Monperrus et al., we only distinguish between these both. In fact, those special cases can be extended to other primitive types as well.

[21] https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html
[22] https://docs.oracle.com/javase/8/docs/api/java/util/logging/Logger.html
[23] https://docs.oracle.com/javase/8/docs/api/javax/crypto/Mac.html
[24] https://docs.oracle.com/javase/8/docs/api/javax/crypto/KeyGenerator.html

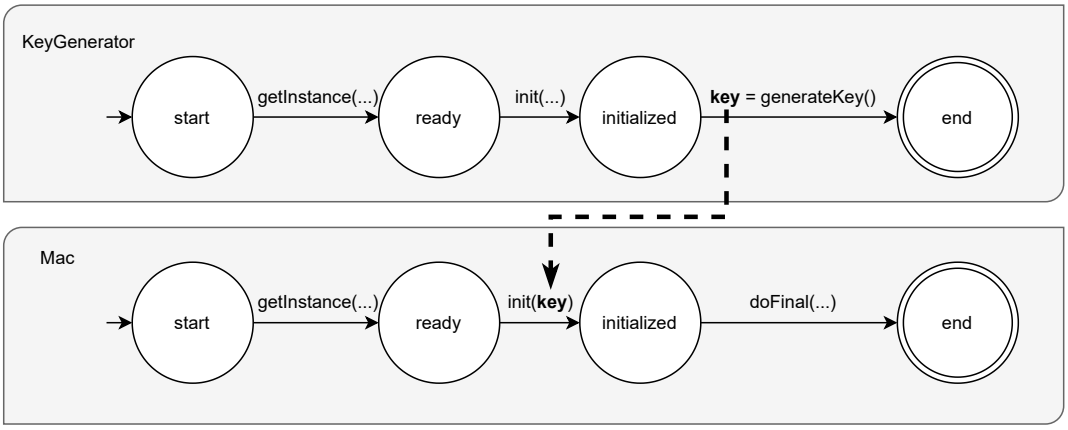

Figure 2.5: Simplified finite-state-machine displaying argument state usage constraint. *javax.crypto.Mac*[4] expects the passed argument *key* to be generated by *javax.crypto.KeyGenerator*[24] through a specific protocol.

String Format constraints specify a restriction for the format of the passed string [METM12]. There are APIs that only allow a certain value from a set of strings. For instance, the method *getBytes(charsetName)* provided by *java.lang.String*[25] returns a byte array depending on the passed charset name. The argument needs to be a valid string from a set of supported charsets (e.g., UTF-8). In total, 2.7% of all analyzed API elements contain such usage constraint [METM12].

Number Range constraints specify that only a set or range of numbers are allowed to be passed [METM12, SSD15]. The prime example is a method that expects a number to be passed in a range of a valid port number identification. In total, 2.7% of all analyzed API elements contain such usage constraint [METM12].

Pre-Null-Check constraints state the requirement to check for null before the argument is passed to the method [METM12, Ama18, Li20, NVN19, SSD15]. These usage constraints rather are — like *Post-Null-Check* 2.4.1.1 — implicit ones. In most cases the documentation states that a *NullPointerException* is thrown when *null* is passed. However, there are also cases where the documentation explicitly restricts to pass *null*. For example, the API class *CollatingIterator*[26] from the *Apache Commons Collection* library exposes the method *addIterator(iterator)*, for which documentation states "[...] the iterator to add the collection must not be null". In total, 13% of all analyzed API elements have at least one such usage constraint [METM12].

Method Parameter Type constraints restrict the passed argument's type [METM12, SSD15]. Although this is contrary to object-oriented programming principles (i.e., generalization [Tai96]), there are cases where this particular usage constraint exists [METM12]. For instance, the *java.sql.Timestamp*[27] API class offers the method *compareTo(date)* which parameter *date* is from type *java.util.Date*[28]. However, the documentation states "Compares this Timestamp object to the given Date, which must be a Timestamp object" [METM12]. In total, 1.9% of all analyzed API elements contain such a usage constraint [METM12].

[25] https://docs.oracle.com/javase/8/docs/api/java/lang/String.html
[26] https://commons.apache.org/proper/commons-collections/apidocs/org/apache/commons/collections4/iterators/CollatingIterator.html
[27] https://docs.oracle.com/javase/6/docs/api/java/sql/Timestamp.html
[28] https://docs.oracle.com/javase/6/docs/api/java/util/Date.html

Method Parameter Correlation constraints restrict the value of passed arguments because of their inter-dependency [METM12]. For instance, the method *setKeyEntry(alias, key, password, chain)* provided by *java.security.KeyStore*[29] requires passing a *key* depending on the provided certificate *chain*. The documentation[29] states: "If the given key is of type java.security.PrivateKey, it must be accompanied by a certificate chain certifying the corresponding public key.". In total, 1.9% of all analyzed API elements contain such a usage constraint [METM12].

2.4.1.4 Multiple Assignments

In addition to API usage constraint types only assigned to one specific part of an API method call (dashed and colored boxes in figure 2.2), there are also usage constraint types assigned to multiple parts we furthermore introduce, namely, *Exception Handling*, *Context* and *High-Level Constraints*. The usage constraint type *Context* is further divided into two subtypes, i.e., *Synchronization* and *Threading*.

Exception Handling constraints impose requirements at the exception-handling level. We first need to distinguish between two conceptual types of exceptions in Java, namely, checked and unchecked exceptions. Checked exceptions must be declared in the *throws* clause behind the method's signature. Thus, the compiler forces the developer to care about the possible thrown exceptions. In contrast, unchecked exceptions do not need to be handled explicitly. A usage constraint of type *Exception Handling* describes the situations in which a developer must consider possible thrown exceptions and must react precisely to them [Ama18, Li20, KTBR20]. Precisely means, for example, not to just catch for all possible thrown exceptions using *Throwable* in the catch block. *Throwable* is the supertype of all exceptions and might suppress other thrown exceptions, which, however, would have to be specifically reacted to individually in order to ensure the frictionless running of the program. Furthermore, when catching *Throwable*, exceptions might also be caught that are not intended to be caught (e.g., Cancellation-Exception that makes it impossible for the user to cancel the command [Ama18]). This type of usage constraint can be associated with all parts of an API usage (cf. white-colored box labeled with *Exception Handling* in figure 2.2). In total, 4.1% of all analyzed API elements contain such a usage constraint [METM12].

Context constraints are related to language constructs that must surround the API usage. Here we do not consider the language constructs that are already claimed by *Exception Handling* and *Controlling Method Call* but all other language constructs surrounding the API usage required by the API. We observed a similar usage constraint type mentioned in previous studies [METM12, Ama18, DH09], but without explicitly limiting the constraint to surrounding language constructs. A prime example for a *Context* constraint is the *java.lang.Object*[30] which requires using its exposed method *wait()* only within loops. The most similar API directive mentioned by Monperrus et al. [METM12] is the *Method Call Visibility Directive* we already introduced in the *Forbidden Method Call* constraint type (cf. section 2.4.1.2). This API directive matches *Context* usage constraints as Monperrus et al. mentioned the visibility of methods related to the context in which they are intended to be used. Therefore, we can report the prevalence as about 4.2%, but rather lower since the API directive overlaps usage constraints of *Forbidden Method Call* and *Threading*. We further subdivide into two more usage constraint types, namely, *Threading* and *Synchronization* (see also grey-colored box labeled *Context* in figure 2.2).

[29] https://docs.oracle.com/javase/8/docs/api/java/security/KeyStore.html
[30] https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html

Threading addresses usage constraints of using an API only on specific threads. The prime example is the *javax.swing*[31] API class, where its documentation states that "[all] *Swing* components and related classes [...] must be accessed on the event dispatching thread" [Orab]. The most similar API directive mentioned by Monperrus et al. [METM12] is the *Method Call Visibility Directive* whereas 4.2% of all analyzed elements contain such a usage constraint. However, as already mentioned in *Context* type, this number is only a rough direction because it overlaps with *Forbidden Method Call* and *Context* constraints.

Synchronization comprises requirements to take care of concurrent access to the API object in a multi-threaded environment (i.e., the requirement of thread-safe use) [Ama18, METM12, KTBR20]. For instance, if at least one thread structurally modifies an instantiation of *java.util.HashMap*[32], there is a need to synchronize the API object externally (e.g., by retrieving a lock). In addition, this type also includes requirements for proper assurance of thread-safe use. For example, a usage constraint of this type is violated if the lock is obtained twice in a nested manner, immediately resulting in a deadlock. In total, 3.9% of all analyzed API elements contain such a usage constraint [METM12].

High-Level Constraints address usage constraints that cannot be covered by the previously presented types of usage constraints in this section. *High-Level Constraints* can cover all areas of an API usage (see dashed box behind all usage constraint types in figure 2.2). For example, the operating system on which the program is to run may be important. Windows machines provide different algorithms for the secure generation of random numbers than do Linux machines [Ste20]. Therefore, this may also affect the interaction with the given API that encapsulates and provide such functionality. Another example are methods that must be invoked within a certain time. We can observe such a usage constraint again in the context of cryptography, where re-keying must be performed at a certain time [Ste20]. In general, we classify such usage constraints as *High-Level Constraints*. Note that none of the studies we considered explicitly mentioned this type of usage constraint, neither did Monperrus et al. [METM12].

2.4.2 Limitations

The work of Monperrus et al. [METM12] and Bruch et al. [BMM10] also considered types of usage constraints related to the extension and inheritance of APIs. Monperrus et al. list these usage constraint types under the main category *Subclassing Directive*. For instance, there is the type *Call Contract Subclassing Directive* that whenever a defined method is overwritten, there is a need to call other predefined methods than just calling the *super()* method. However, our classification framework (cf. section 2.4.1) only comprises usage constraint types that restrict an API's usage (i.e., not at extension or inheritance). This is due to the specification language CrySL which focuses on the specification of usage constraints related to the actual use of cryptographic APIs and not on inheritance or extension. Therefore, we want to align to the same application context.

2.4.3 Overlapping Characteristics

Although we provide a more fine-grained classification framework than recent studies in addition to an association of usage constraint types related to the actual use of an API — there are cases where the types are not orthogonal to each other. For example, this is the case when we consider *Pre-Null-Check* 2.4.1.3 and *Exception Handling* 2.4.1.4 constraints. An API can indicate that

[31] https://docs.oracle.com/javase/8/docs/api/javax/swing/package-summary.html
[32] https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html

a *NullPointerException* is thrown when null is passed to a method call. However, this can be seen as a usage constraint of *Pre-Null-Check* (i.e., checking the argument for not being null before it gets passed to the method call). Conversely, it can also be seen as a usage constraint of *Exception Handling* because if it is not checked in advance, the developer must consider the possible thrown exception.

2.5 Discussing Differences of Classifications

The most similar classification framework compared to our framework introduced in section 2.4 is *MuC* elaborated by Amann [Ama18] and its refined version provided by Li [Li20] (cf. section 2.1.3). Both studies introduced an exhaustive classification framework of API misuse types and indicated the prevalence of each elaborated type. Amann's study is based in part on the findings of Monperrus et al. [METM12], who examined the different types of API directives, including API usage constraints. Because Li furthermore used the work of Amann as a basis to refine their framework, they also implicitly used the findings from Monperrus et al. Thus, their classification frameworks are comparable to ours as we also take the findings of Monperrus et al. as a basis. Although our classification framework is on the level of API usage constraint types, we can compare our classification framework to theirs as a violation of an API usage constraint is an API misuse (cf. definition 2.4). Nevertheless, they have a slightly different view regarding API misuses, which we will discuss next. In this section, we show that our classification framework is at least as expressive as theirs. Moreover, our classification framework provides a more fine-grained classification of API usage constraint types — and API misuse types. For the sake of clarity, we annotate Amann's introduced types with "A", Li's with "L", and ours with "S" respectively.

API-Misuse Classification (MuC) [Ama18]. The API-Misuse Classification framework (MuC), introduced by Amann, encompasses four main categories: Method Calls, Condition, Exception Handling, and Iteration. Although every main category has its subcategories, we only introduce subcategories' details when there are (i) clear differences between the violation of our introduced usage constraint types or (ii) this subcategory does not refer to API misuse based on our definitions.

The category **Method Calls$^{\mathbf{A}}$** classifies misuses on API method calls that are not called in the correct place. This includes *Missing Method Calls*^{A}, i.e., a method is not called even though the API requires to call the particular method, and conversely *Redundant Method Calls*^{A} that are called even though the API restricts the invocation. This category is covered by the violation of *Method Call Sequence*^{S} as its constraints define the allowed and required method calls based on the API object's state.

Their category **Conditions$^{\mathbf{A}}$** classifies misuses that target method calls even though required conditions are not fulfilled in advance. This also includes violations of the requirement to value and state conditions of passed arguments; for example, the passed string is not in a certain format. Here, subcategories are *Missing Null Check*^{A} and *Redundant Null Check*^{A}. *Missing Null Check*^{A} is the violation not to check for null before an argument is passed to the API call, although the API requires it to be checked. This also includes violations where the API method returns a possibly null value and the receiver is not checked. We split this category up into *Pre-Null-Checks*^{S} — the violation of the usage constraint to not pass a null value as an argument —, and *Post-Null-Check*^{S} — the violation of the usage constraint to not check the receiver for null. This is more intuitive because it separates this type of API misuse into the different parts of API use. The other subcategory, *Redundant Null Check*^{A}, includes null checks at an improper place, for instance, when a null check is made after the respective method is called (see figure

```
public void misuse() {
  API o = new API();
  API a = o.retrieveAPI(); // may return null
  a.doSmth(); // usage without checking for null
  if(a == null) { // late null-check
    a = new API();
  }
}
```

Figure 2.6: An example in the subcategory *Redundant Null Checks*^{A} from *MUBench Dataset* [ANN+16] that shows a null check which is performed after the method call. However, it should be checked before the method call. This misuse is classified as a violation of *Post-Null-Check*^{S}.

```
public void noMisuse(API a) {
  int val = o.useAPI(); // never returns null
  if(val != null) {
    val.goOn();
  }
}
```

Figure 2.7: Another example in the subcategory *Redundant Null Checks*^{A} that shows a null check which is not needed because *useAPI()* never returns null. This is no violation of an API usage constraint, and thus, no API misuse by our definition.

2.6). On the other hand, it also covers cases where a null check is made that is actually not needed (see figure 2.7). The former example (figure 2.6) is covered by the violation of *Post-Null-Check*^{S} because the null check is simply missing before further usage. The latter example (figure 2.7) could be considered a misuse because of the unnecessary check for null. Although it is a non-optimal way to use the API, it does not cause any error. Furthermore, the API usage constraint to check the return value for null is not imposed by an API designer/expert. Hence, we do not consider it a misuse (cf. definition 2.4).

The subcategory *Missing Value and State Condition*^{A} aims at API misuses regarding the state of the API object: a method is called even though the state of the API restricts the usage, and at format and state of passed arguments — for example, the passed number is not in the range required by the API. This subcategory is — indeed — very broad and actually comprises many kinds of usage constraint violations, i.e., constraints of type *Method Call Sequence*^{S}, *Controlling Method Call*^{S}, and all the subtypes listed in *Passed Arguments*^{S}, excluding *Pre-Null-Check*^{S}, and *Argument Type*^{S}. Thus, our separation allows us to assess API misuses more fine-grained. *Redundant Value and State Condition*^{A} aims at the use of unnecessary conditional checks before using an API method, e.g., to check a collection for items even though there is a guarantee that an item was added in advance. Indeed, this can be seen as a misuse because unnecessary conditional checks result in a non-ideal API usage but do not negatively affect the program flow. Thus, according to definition 2.4 they are considered not as an API misuse. The only classified API misuse (cf. figure 2.8) in the subcategory *Redundant Value and State Condition*^{A} from the MUBench Dataset [ANN+16] shows an iteration through a list of strings. Because the variable i is used counting from one (to the length of the list), and inside the for loop, we access the items through *get(i - 1)*, the last element of the list is not accessed. However, this is a misuse only from a developer's perspective, since such a usage constraint is imposed in their development context. From the perspective of an API designer/expert, it is not an API misuse (cf. definition 2.3) because the API does not require fetching all elements from the list.

```
public void noMisuse(List<String> l) {
  for(int i = 1; i < l.size(); i++){
    l.get(i - 1); // the last element of list l is not considered
  }
}
```

Figure 2.8: Example in the category *Redundant Value and State Condition*^{A} where the last element of the list is not retrieved. This is no API misuse as the usage constraint is imposed by the developer, not by the API designer/expert.

The subcategory *Missing Synchronization Conditions*^{A} classifies API misuses that imply an improper use of the API object regarding synchronization — for example, in a multi-threaded environment where the API object is used without first obtaining a lock. This covers exactly violations of *Synchronization*^{S} usage constraints. In contrast, *Redundant Synchronization Conditions*^{A} classifies API misuses for two typical cases. The first case describes situations in which a lock is obtained unnecessarily. The second case aims at situations where a lock is obtained twice in a nested manner, leading immediately to a deadlock. According to our definition, the first case is no API misuse if and only if it does not negatively affect the further frictionless program flow (cf. definition 2.4). The second case is covered by the violation of *Synchronization*^{S} constraints.

Finally, their subcategory *Missing Context Conditions*^{A} aims at API misuses related to threading. For example, GUI components from SWING[33] need to be accessed on the event dispatching thread; if not, then it is accordingly an API misuse. These kinds of scenarios are covered by the violation of *Threading*^{S} constraints. Moreover, we provide a more fine-granular view as our type *Context*^{S} also comprises constraints related to language constructs that need to surround the API usage, which Amann does not directly consider. The other subcategory *Redundant Context Conditions*^{A} classifies API misuses related to threading whose usages are merely redundant and therefore not needed. For example, when a GUI element dispatches work on a UI element to the dispatching thread while the current execution context is already on the dispatching thread. If such a non-ideal API usage leads to errors or misbehavior of the API, it is classified as the violation of *Threading*^{S} in our framework.

The category **Iteration**$^\mathbf{A}$ aims at violations of APIs like collections, iterators, or IO streams, particularly with loops and recursive methods. Although the use of control methods used in conditions are frequently used with such APIs, misuse of such would be not to take those into account. For example, before we can retrieve the *next()* element from an iterator, there is a need to check whether it *hasNext()* element(s). In fact, these cases can be seen as a violation of *Controlling Method Call*^{S} constraints and not as a separately listed type.

Finally, the **Exception Handling**$^\mathbf{A}$ categories target the violation of improper exception handling when using an API. This category matches our definition of the violation of *Exception Handling*^{S} constraints with one exception: unlike *Exception Handling*^{A}, *Exception Handling*^{S} does not include violations of usage constraints related to method calls. There is a possibility that a usage constraint on a method call is violated because of a beforehand thrown exception. For example, consider figure 2.9. In case of a possible thrown exception, when *read()* is executed, the prior opened resource is not closed properly. The correct implementation is to *close()* the resource in a finally-block. However, this API misuse is not related to improper exception handling; it is rather a non-assurance that *close()* gets invoked and thus, classified as a violation of *Method Call Sequence*^{S}.

[33] https://docs.oracle.com/javase/8/docs/api/javax/swing

```
public void noExceptionHandlingMisuse() {
  FileReader r = new FileReader("file"); // file exists
  try {
    r.read(); // may throw exception
    r.close(); // not executed in case of a thrown exception
  } catch(IOException) { /** recover **/ }
}
```

Figure 2.9: Example in the category *Exception Handling*^{A} that is not categorized in the category of *Exception Handling*^{S} but in *Method Call Sequence*^{S}. *FileReader* requires to call *close()* at the end of its usage. However, *close()* may not be called in the case of a thrown exception executing *read()*.

Refinement of MuC [Li20]. Following the study by Amann [Ama18], Li [Li20] did a more exhaustive and wide-ranging empirical study on API misuses in the wild in 2020. They restructured and refined the classification framework of Amann and introduced new subtypes in addition to *Method Calls*^{A}, namely, *Replaced Arguments*^{L}, *Replaced Name*^{L}, *Replaced Name and Arguments*^{L} and finally, *Replaced Receiver*^{L}. These additional types result from their technical analysis of API misuses in bug-fixing commits. They compared the different revisions a commit contains (i.e., the respective code before and after the change). Thus, they found that there are bug-fixing commits that only consist of the change that is related to the order of passed arguments to an API method call (*Replaced Arguments*^{L}), the method name (*Replaced Name*^{L}) or both *Replaced Name and Arguments*^{L}. However, there were also cases where the whole API class or object on which the method is called was changed (*Replaced Receiver*^{L}). These newly introduced subtypes can be covered by ours listed in *Method Calls*^{S} and *Passed Arguments*^{S}.

In summary, our classification framework provides at least as much expressiveness as *MuC* [Ama18] and Li's [Li20] refined framework. Moreover, we provide a more natural view on API misuses by classifying usage constraint types (i.e., the root cause of an API misuse) instead of classifying API misuse types. We also provide a more fine-grained view of usage constraints. Our classification framework includes the API usage constraint types *Method Parameter Type*^{S}, *Method Parameter Correlation*^{S} and *High-Level Constraint*^{S}, which are not included in *MuC* nor in the framework of Li. In addition, our classification framework has split individual types to better locate them based on an API usage. For example, *Missing Null Check*^{A} is further distinguished in *Post-Null-Check*^{S} (i.e., required null check on the receiver) and *Pre-Null-Check*^{S} (i.e., required null check on the parameter), which comes closer to the actual use of an API method call.

However, we also showed the slightly different view on API misuse types. *MuC* classifies a simple redundant use of an API without further negative impact on the program flow as API misuse. In contrast to our definition (cf. definition 2.4), these cases are not classified as an API misuse. Also, Li introduced API misuse types resulting from their technical investigation of API misuses: an API misuse is considered to be a change operation in a bug-fixing commit related to an API. For example, an API misuse is classified as *Replaced Arguments*^{L} when an API method call occurs in a bug-fixing commit where the only change to the method call is the reordering of the arguments. All in all, we argue that our definitions (cf. section 2.3), as well as our classification framework (cf. section 2.4), provide a more natural and fundamental view of API misuses and their origins. As a result, this answers our second research question (**RQ2**) for a classification framework that can be used to evaluate the capabilities of a specification language like CrySL and an analysis tool like CogniCrypt precisely.

2.6 Prevalence of API Usage Constraints and its Violations

In this section, we provide an approximate overview of the prevalence of the API usage constraints listed in our classification framework (cf. section 2.4) and their violation. Therefore, we take the empirical studies from Monperrus et al. [METM12], Amann [Ama18] and Li [Li20] as a basis. Monperrus et al. conducted an empirical study of the prevalence of API directives in API documentation by analyzing API elements (e.g., classes, methods and fields) from three libraries, namely *Java Class Library*[34], *JFace*[35] and *Apache Commons Collection*[36]. Amann conducted an empirical study of API misuse types in bug datasets based on their classification framework, *MuC*. Followed by the study of Amann, Li [Li20] conducted an empirical study of API misuse types in the wild by analyzing all bug-fixing commits on GitHub in the timeline between 2011 and 2018.

However, as discussed in the previous section, Amann and Li have a slightly different view on API misuses. Also, Monperrus et al. analyzed API elements with the focus on API directives, which do not necessarily need to be API usage constraints (cf. definitions 2.2 and 2.3). Therefore, we can only provide approximate numbers for each of our API usage constraint types. Nevertheless, we matched exactly those types to ours that fit our definition of API usage constraints and API misuses. The exact mapping of our and their types can be seen in appendix 8.1 whereas the approximate overview of each types' prevalence can be seen in table 2.1. There are two reasons (i-ii) for the approximation:

(i) Due to our more fine-grained classification and slightly different definitions, some types may overlap. Thus, we can only give approximate numbers in table 2.1 that result from the following explained cases:

> **Less than or equal to** indicate that we can only state that the prevalence can be up to the given number. For example, the type *Missing Value and State Condition* introduced by Amann covers most of our types under *Passed Arguments*. Consequently, we can only list the number of *Missing Value and State Condition* to each of our types under *Passed Arguments*. Moreover, *Missing Value and State Condition* also classifies API misuses beyond misuses related to passed parameters. It also covers our type *Controlling Method Call*. Thus, we can only state that the prevalence must be equal to or less than the given number.
>
> The **range** results from the prevalence, where one of their types exactly matches our type plus types that only partly match. For example, the type *Method Call Sequence Directive* introduced by Monperrus et al. matches exactly our definition of *Method Call Sequence*. In addition, Monperrus et al. introduced the type *Miscellaneous Method Call Directive*, which classifies directives related to method calls but are somewhat different and so rare that no separate type has been introduced. Thus, the exact match of *Method Call Sequence Directive* defines the lower bound, the addition of both the upper bound. This means the prevalence is somewhere between.
>
> **Not available (n.a.)** results from types where we know from other work that those usage constraints exist but have not been further investigated in previous empirical studies and for which we have no figures. These types are also not further considered in the summary of the associated API usage part; for example, the given number of *Receiver* only consists

[34] https://docs.oracle.com/javase/8/docs/api/allclasses-frame.html
[35] https://wiki.eclipse.org/JFace
[36] https://commons.apache.org/proper/commons-collections/

of *Post-Call(s)* in the case of prevalence for API documentation. Nevertheless, the figures will likely be higher.

(ii) The introduced types from Amann and Li are not orthogonal to each other. We can observe this in the *MUBench Dataset*, where one API misuse is classified by many types, and Li's refined classification framework, where they explicitly mention that types can overlap. Therefore, the figures can be scattered.

Table 2.1 shows an overview of the prevalence for each API usage constraint type of our classification framework (left column). The boldly highlighted rows are the parts of an API usage and group API usage constraint types associated with it. For example, *Post-Call(s)* and *Post-Null-Checks* are associated with the receiver of an API usage. Thus, the row *Receiver* adds up the prevalence of all its associated API usage constraints and API misuses.

The shown percentage of each type in the columns labeled with *API Misuses* indicates the fraction measured against the total amount of API misuses found. This has to be taken into account because we cannot simply say that the highest percentage of API misuse also means that this type is particularly vulnerable to the violation of such usage constraints. We can only claim that a particular type is most susceptible to API misuses if we measure the occurrence of API misuse against the total amount of API uses concerning that particular constraint type. For example, if we could determine that there were a total of 50.000 API uses that were subject to a particular API usage constraint type, and 10.000 of those were misused, which would yield

Table 2.1: Approximate prevalence of API usage constraints and its violations, based on empirical studies from Monperrus et al. [METM12], Amann [Ama18] and Li [Li20].

API Usage Constraint Types	**Prevalence of**		
	API Usage Constraints	**API Misuses**	
	Monperrus et al. (%)	Amann (%)	Li (%)
Receiver	**0.9**	**≤ 27.8**	**≤ 1.4**
Post-Call(s)	0.9	n.a.	n.a.
Post-Null-Check	n.a.	**≤ 27.8**	**≤ 1.4**
Method Call	**6.2 – 16.4**	**51.0 – 70.2**	**≤ 54.0**
Method Call Sequence	6.2 – 12.2	22.7 – 32.8	≤ 48.4
Controlling Method Call	n.a.	28.3 – 28.7	≤ 5.7
Forbidden Method Call	≤ 4.2	≤ 8.6	n.a.
Passed Arguments	**22.2**	**≤ 56.8**	**≤ 15.0**
Argument State	n.a.	≤ 28.8	≤ 5.7
String Format	2.7		n.a.
Number Range	2.7		
Argument Correlation	1.9		
Argument Type	1.9	n.a.	
Pre-Null-Check	13.0	**≤ 27.8**	≤ 1.4
Multiple Assignments	**4.1 – 16.4**	**8.6 – 13.6**	**2.2 – 2.3**
Exception Handling	4.1	8.6	2.2
Context	≤ 4.2	≤ 2.0	n.a.
Synchronization	3.9	0.5 – 1.0	0.04 – 0.05
Threading	≤ 4.2	≤ 2.0	n.a.
High-Level Constraints	n.a.	n.a.	n.a.

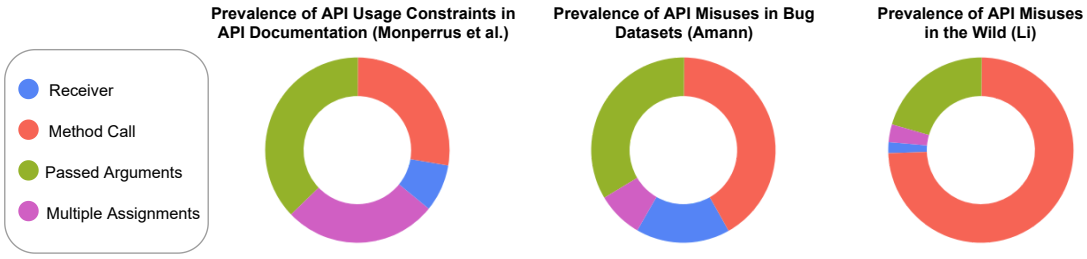

Figure 2.10: Visualization of the prevalence of API usage constraints in API documentation [METM12] (left-hand chart), the prevalence of the violations of API usage constraints (i.e., API misuses) in bug datasets [Ama18] (chart in the middle) and in the wild (right-hand chart) [Li20]. Visualization is based on table 2.1.

20% of misuses in that particular type. This would enable us to claim that developers heavily struggle with *Synchronization* usage constraints because of the high fraction of API misuses. However, this is not the case in Amann and Li's studies as they calculate the fraction of each type measured against the total amount of API misuses found. Therefore, it may be that APIs with underlying usage constraints, for example, *Synchronization* constraints, are used less, and therefore, the number of API misuses is lower. Nevertheless, as an orientation, we can use the column *API Usage Constraints* (i.e., the findings of Monperrus et al.) related to the occurrences of API usage constraints in API documentation.

We can observe API usage constraints of type *Method Call Sequence* are quite common in API documentation ($6.2 - 16.4\%$) and are overall the most violations in bug datasets as well as in the wild ($\leq 54.0\%$). The least amount of API misuses are violations of usage constraints related to *Synchronization* ($0.5 - 1.0\%$ in case of Amann) and ($0.04 - 0.05\%$ in case of Li). The largest percentage difference in API misuses is in type *Passed Arguments* with $\leq 56.8\%$ in bug datasets and $\leq 15.0\%$ in the wild. This may be due to the uninvestigated violations of API usage constraints *String Format*, *Number Format*, *Argument Correlation* and *Argument Type* by Li.

For ease of comparison, the prevalence of each boldly highlighted row is shown in a pie chart in figure 2.10. Interestingly, API usage constraints related to the method call are typical in API documentation (left-hand pie chart) and are overall the most misused in bug datasets and in the wild (middle and right-hand pie chart). API usage constraints with multiple assignments such as *Exception Handling* are frequent in API documentation but are somewhat less misused.

However, although the violation of *Exception Handling* constraints is lesser in both empirical studies (8.6% and 2.2% respectively), they should not be neglected. Correct exception handling is mandatory for the software's frictionless running, especially when there is no way to check compliance with usage constraints in advance. For example, the *java.util.regex.Pattern*[37] API class provides the method *matches(regex, input)* that attempts to match the regular expression (regex) against the given input. It is required to pass a valid regular expression. Thus, the method enforces a usage constraint of type *String Format*. If an invalid regular expression is passed, an unchecked *PatternSyntaxException* is thrown. The passed regular expression can be checked for compliance with the *compile(regex)* method, which is also used when the regular expression is simply passed to *matches(regex, input)*. Thus, compliance with the enforced usage constraint *String Format* is ensured by correct exception handling. This fact becomes even clearer when we consider that proper exception handling is associated with all parts of an API use. Thus, proper exception handling can ensure compliance with other usage constraints (see also section 2.4.3).

[37] https://docs.oracle.com/javase/8/docs/api/java/util/regex/Pattern.html

3 API Misuse Detection

API misuses are a common problem when using third-party software, as discussed in chapter 2. Therefore, several approaches have been developed in the past to mitigate API misuses. We can fundamentally distinguish such approaches in two areas [Ama18].

The first area includes approaches that actively try to push the knowledge of usage constraints to the attention of developers [DH09]. Because it can be tedious to extract important information from the underlying documentation of an API, developers can be made aware directly in the documentation about API usage constraints by highlighting them. Since documentation may not be complete, contradictory, or is even incomplete or missing, this approach may not be enough to mitigate API misuses. On the other hand, some approaches try to recommend the correct usage directly in the development environment [BMM09]. Nevertheless, while these approaches can increase the awareness of proper API uses, they do not prevent or uncover API misuses.

The second area classifies approaches that detect API misuses in a given codebase, referred to as *API misuse detectors*. Detecting API misuses is a classic problem of software engineering [LWY+18]. In the past, several approaches have been developed to detect API misuses. In addition to approaches that attempt to detect general API misuses, some approaches are tailored to specific domains, such as cryptography [RXA+19, Ste20]. In the following, we focus on API misuse detection tools. Therefore, in section 3.1, we first introduce and classify different types of approaches providing a broad overview. Also, we explain the basics of static analysis (section 3.2). Overall, this helps to understand the approach of CrySL and CogniCrypt which we will introduce in section 3.3 and 3.4, respectively.

3.1 Types of API Misuse Detection

The different types of API misuse detection approaches vary, as shown in figure 3.1. Nevertheless, all approaches have the same goal: detecting and indicating API misuses with high precision and high recall. High **precision** means that the fraction of true API misuses (true positives) among all reported API misuses is high. Indeed, there may be the case that an API misuse is indicated that is actually no API misuse (false positive). High precision is a desirable property as studies have shown that developers tend not to use API misuse detection tools if they indicate many false positives and do not work accurately [JSMB13]. In fact, studies have also shown that many of the available API misuse detection tools suffer from a relatively low precision rate [Li20]. On the other hand, high **recall** means that a high fraction of API misuses from a set of all existing API misuses in a program is found.

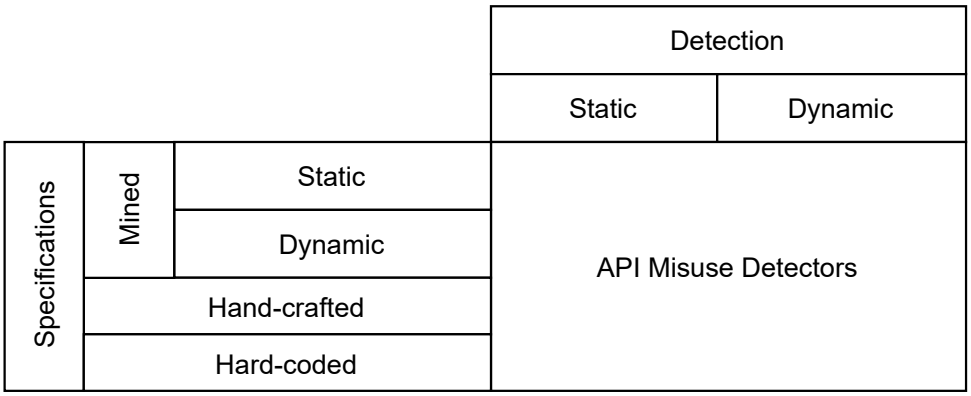

Figure 3.1: Overview of different API misuse detection approaches. This visualization is based on Amann's [Ama18] figure with the difference that also detection tools using hand-crafted or hard-coded specifications are API misuse detectors.

The detection of API misuses can be performed in two different ways: static or dynamic analysis (upper right-hand part of figure 3.1). The analysis is in both ways conducted based on specifications (lower left-hand part). These specifications can be statically or dynamically mined, hand-crafted in a specific format, or hard-coded into the respective API misuse detector.

In the case of **static analysis**, the source code or binaries are examined for API misuses without being executed [Ama18]. In the case of binaries, such as Java Bytecode, the code must at least be compilable. This is advantageous compared to dynamic analysis techniques because we can conduct the static analysis, although the program might not be ready to be executed. As a result, API misuses can be found in a very early stage of the development process [ASN17]. Besides, the detection of an API misuse is made on a relatively close representation of the source code. This enables, for example, the indication of an API misuse directly in the development environment (i.e., in the source code) [Ste20]. An example for a static analysis tool is CogniCrypt [KNR+17]. We provide a more detailed view on static analysis in section 3.2.

Unlike static analysis, **dynamic analysis** requires the program to be executed [Ama18]. Therefore, the common technique is to analyze the program through execution traces (i.e., a finite sequence of events, for example, method calls) or logs. Thus, most of the dynamic analysis detection techniques rely on test case generation, i.e., the generation of sufficient test cases to reach an overall high trace coverage. For example, manually created training datasets or randomized test case generation can be used for this purpose. Depending on the quality of the training dataset, there may be execution traces that are not captured. As a result, the precision of the detection tool suffers. In contrast, this may also be an advantage as only those program traces are examined that can actually be reached [HP04]. By not reaching infeasible traces, dead code, for example, is not considered. Another problem is that most dynamic detection techniques do not examine usage elements other than method calls [Ama18]. With that said, primarily violations of API usage constraints related to a method call are analyzed. Other violations of usage constraints like *Passed Arguments* (cf. section 2.4.1) are not taken into account, resulting in a likely high number of undetected API misuses if we consider the prevalence of API misuses once again (cf. section 2.6). They are usually only implicitly recognized if the respective API misuse causes an exception. Thus, an API misuse is detected in the sense that it is problematic due to a thrown exception. Overall, exception detection allows a significant reduction of false positives compared to static analysis techniques [Ama18]. The dynamic analysis approach presented by Pradel and Gross [PG12], for example, is characterized by the fact that it shows no false positives but only true positives when applied to ten well-tested

Java applications.

Both, static and dynamic analysis techniques, have one common problem: they need to know the correct (or incorrect) use of an API. Therefore, they need specifications (lower left-hand part of figure 3.1) that describes the correct (or incorrect) use of an API. These specifications can be obtained in three different ways: by statically or dynamically mining them or by providing hand-crafted specifications.

Statically mined specifications are inferred automatically by algorithms on (representations of) source code [Ama18]. Such a statically mined specification approach is, for example, implemented in the static analysis tool *Jadet* [WZL07]. The specification is gathered when a certain amount of usages indicates the correct usage, i.e., a common pattern is found. Every deviation from that common pattern is then considered as an API misuse [Ama18]. In the case of statically mined specifications, the common usage patterns are mined on already written source code (or binaries). Therefore, the project itself can be used as a source for the mining technique, or the mining is conducted in a large-scale manner (e.g., by using projects from GitHub). Another approach is to mine the specification from an API's underlying documentation, code-search engines, or Q&A sites. Statically mined specifications have the advantage that they can be created quickly, reproducible, and targeted. However, when extracting specifications from documentation, the same problems arise as already mentioned in the definition of API usage constraints (cf. section 2.3), i.e., the documentation may not necessarily be complete, may be contradictory or missing, or the API takes domain-specific knowledge for granted. Mining specifications from code (or binaries) has the disadvantage of mining first an exhaustive amount of correct usage patterns to then define the correct usage. This is not a trivial task, for example, if we consider APIs that are newly introduced [WLW+19]. An insufficiently large and qualitative amount of code (or binaries) containing the desired usage patterns is one of the main reasons for the large number of false positives [Ama18, NHKO20].

Dynamically mined specifications are inferred automatically by looking at the execution traces (or logs) of a program [Ama18]. Just like statically mined specifications, there is a need to have an exhaustive large amount of execution traces that manifest the correct usage pattern. Such a dynamically mined specification approach is, for example, implemented in *RV-Monitor* (Runtime Verification Monitor) [LZL+14].

When mining specifications from code, static and dynamic mining approaches are based on a decisive assumption: the majority of usages is correct, and less frequent usages may be API misuses [Ama18, Li20, WLW+19]. However, this assumption does not necessarily have to be true in practice since a rare usage does not necessarily mean that it is an API misuse.

Therefore, **hand-crafted specifications**, created by domain experts or API designers, have the advantage of defining the real correct (or incorrect) usage of an API. The huge downside is that hand-crafted specifications are costly, error-prone, and need to be maintained from time to time. For example, CrySL [KSA+18] is a domain-specific specification language to specify API usage constraints for cryptography-related APIs.

There are also approaches shipped with **hard-coded specifications**, which are not editable by the user of the API misuse detector. For example, *CryptoLint* [EBFK13] uses six hard-coded rules to check for cryptographic API misuses.

3.2 Basics of Static Analysis

Static analysis is performed on the source code or binaries directly, and therefore, no execution of the program is needed. Because static analysis is performed in a goal-driven manner, it abstracts from the source code or binaries to focus on the relevant information needed. Overall, it simplifies the analysis to such an extent that only those language constructs are mapped

which are necessary for an analysis of the program [Ste20]. This abstraction layer, also called **intermediate representation** (IR), is then used to conduct the actual analysis. Besides, the code or binary files are transformed into other models, such as *control-flow graphs*.

Control flow graphs (CFGs) represent statements of the original program code as nodes, and potential control-flows like branches or loops as edges [Ste20]. A static analysis taking the control flow of a method and the order of statements into account is considered **flow sensitive**. In general, a distinction is made between intraprocedural analysis, where only instructions residing inside a single method are considered, and interprocedural analysis, which considers statements beyond the boundaries of a single method. An interprocedural analysis is called **context sensitive** if it takes the calling context into account. In other words, when the analysis of a called method is finished, the static analysis can return to the original call-site.

In contrast to CFGs, **call graphs** are used to visualize the connection of caller and callee relationships [Ste20]. Unlike CFGs, call graphs model method calls as nodes and connections as edges. However, there is also the possibility to combine both approaches, CFGs and call graphs, as so-called **interprocedural control-flow graphs** (ICFG).

The representation of program code in the form of graphs enables different types of analyses, for example, **typestate analysis**. A typestate defines the allowed operations that may be performed on an object at a given time [Ste20]. This type of analysis can be used to check for compliance concerning an API object's underlying state-machine, and thus, compliance with a given usage protocol (cf. API usage constraints of type *Method Call Sequence* in section 2.4.1.2). To support a typestate analysis, it is likely to conduct a points-to analysis as well [Ste20]. For example, consider a method that accepts one parameter specified as p. Assume that the method is used several times, each invocation with a different object passed. There may now be the need to analyze p to figure out its origin (i.e., to which allocation site p may point to). An **allocation site** is representative for an instruction that creates a runtime object [SB15]. With a points-to analysis, a set of all possible allocation sites is compiled a variable may point to — also called points-to set [Ste20].

The static analysis is **path-sensitive** when the analysis considers conditions that affect a program to branch [Ste20]. However, path-sensitivity is not a trivial task, as it is a statically undecidable problem. Since the decision of a program to branch can be, for example, dependent on user input, it may be impossible to analyze which path is taken.

3.3 CrySL

Many studies have shown that developers often fail to integrate cryptographic APIs in code securely [EBFK13, LCWZ14, CNKX16]. Nadi et al. [NKMB16] were the first who investigated this problem space and found that developers find cryptographic APIs hard to use because of insufficient documentation without examples, the inadequate API abstraction level, and the overall API design. While a re-design of APIs may be a solution in long-term to address these shortcomings, other solutions are needed to detect and fix existing insecure API uses. However, there are approaches to help developers find insecure API uses, but they rely primarily on lightweight syntactic checks generating a high number of false positives. Furthermore, they are based on hard-coded cryptography-specific rules, which prevents extensibility to other APIs (cf. figure 3.1).

Therefore, CrySL was invented to provide cryptography experts an expressive and easy-to-use language to define the correct and secure use of cryptography-related APIs [KSA+18, Ste20]. Consider the figure 3.1 of API misuse detector types again. We can classify CrySL as a specification language to define hand-crafted specifications. These specifications are then consumed by CogniCrypt to conduct a static analysis to detect cryptography-related API

misuses. We will discuss CogniCrypt in detail in section 3.4. This is different from other comparable API misuse detectors, as they mostly mine their specifications from a certain amount of correct usages [Ama18] (cf. section 3.1). Nevertheless, CrySL was not solely developed to serve as a specification for static analysis purposes. It can also serve as documentation, patch generation, or use-case-based code generation [Ste20]. Moreover, it is not tied only to be used by CogniCrypt. It is a stand-alone language that other detectors can also use. CrySL is not tied to specifications only in the field of cryptography, as there are no limitations here. Nonetheless, it is considered a domain-specific language because it was originally intended for specifications in the field of cryptography, and thus, was tailored for this purpose [Ste20]. In the following, we introduce CrySL by referring mainly to the work of Krüger [Ste20]. Whenever we introduce mathematical variables, which are referenced repeatedly throughout the whole chapter, we highlight them for the sake of readability.

3.3.1 Design Goals

CrySL is intended to help API designer/experts of cryptographic APIs to define the correct and secure use of their APIs, bridging the gap between the users and API designer/experts (cf. section 2.1.3). Based on the experience of cryptography experts, the following design decisions were made.

White Listing. Because cryptographic APIs have the property of being used correctly and securely only in a few ways, but in contrast to that, they can be misused in many ways, the authors decided to focus mainly on a whitelisting strategy. That is, defining only the correct usage, rather than defining all the possible ways the API can be misused.

Focus on domain-specific Usage Constraints. According to Krüger, many API misuses are related to data flows and typestate properties. Data-flow properties are consulted when the analysis targets the proper creation and use of data. Some cryptographic APIs encompass usage constraints on passed arguments, such as a properly randomized byte array. On the other hand, there is a need to specify typestate properties that aim at the allowed, forbidden, and required method calls at a given state of the API object. Because in the context of cryptography, it is likely to compose several method calls in a particular order, also between different APIs, to achieve a specific goal, CrySL needs to provide sufficient expressiveness to define such.

Tool independent semantics. CrySL was developed with the requirement of tool independence in mind. This requirement elevates CrySL from being used only by CogniCrypt to a special-purpose language for defining specifications for cryptography-related APIs.

Simplicity. CrySL was developed to be used by cryptography experts. CrySL is therefore tailored to their needs, intuitive to use, and easy to learn.

3.3.2 Capabilities of CrySL explained by an Example

A CrySL rule is separated into smaller sections according to the separation-of-concerns paradigm [DWPJV02]. For example, the specification of a required method call sequence and forbidden method calls are separated into two distinguishable sections, beginning with the keywords `ORDER` and `FORBIDDEN` respectively. This separation increases the readability, re-usability, and clarity of a CrySL rule. Next, we explain each section by an example rule that specifies the correct

and secure use of *javax.crypto.Mac*[1] API class, displayed in figure 3.2. Note that we have not presented the rule in its entirety and have only included the necessary parts for the explanation.

After the keyword **SPEC**, the reference type T is specified, i.e., the fully-qualified API class name which is *javax.crypto.Mac* (line 1) in the example. This section connects the CrySL rule to runtime objects of type T in the client's application code.

The section **OBJECTS** contains typed variable declarations $v \in \mathbb{V}$ where each v is represented by the syntax *TYPE varname*. If the variable is not a primitive type, the fully-qualified class name is required. The specified variable declarations are typically used as parameters in further sections. For example, *java.lang.String macAlgorithm* (line 8) is specified to be bound to the parameter of the method *getInstance(macAlgorithm)* in line 18.

Within the section **FORBIDDEN**, forbidden method calls can be defined, i.e., methods that are not allowed to be called over the whole lifetime of the API object. These forbidden method calls are specified by their method signature f where $f \in \mathbb{M}$. $\mathbb{M}$ is the set of all resolved method signatures, including argument types (i.e., the resolved method signatures depending on T). Argument types need to be specified with its fully-qualified class name (e.g., *methname(java.lang.String)*). The `FORBIDDEN` section is optional. Although specifying forbidden methods deviates from the whitelisting approach, this section is necessary due to the design goal of simplicity. If there were no `FORBIDDEN` section, then all allowed and required methods would have to be listed in the `ORDER` section, except for the forbidden method. This would unnecessarily inflate the specification with the consequence of illegibility. We further refer to $F \subseteq \mathbb{M}$ as a set of all forbidden method signatures.

The section **EVENTS** allows to specify several method event pattern p of the form (m, v) where m is the method signature from $\mathbb{M}$ and v are bound variables from $\mathbb{V}^*$. The specified method event patterns are needed to define the order in which methods are allowed (and required) to be called (i.e., the method call sequence). Each event pattern p is prefixed with an individual label which simplifies the reference of event patterns in the `ORDER` section. We further denote the set of all methods referenced in `EVENTS` section by M. We need to clarify three peculiarities here. Firstly, a variable $v \in \mathbb{V}$ can be bound to the return value or parameters of the specified event pattern p (i.e., the method signature). This is needed if the return value or parameters need to be taken into account for further action. For example, the return value of *doFinal(input)* is bound to the variable *output1* in line 31, which is further used in the `REQUIRES` section. Secondly, method names can be declared twice due to method overloading, each with different parameter types. In some cases, a method is overloaded several times with only changing one parameter at a place; there is the possibility to substitute such a parameter with an underscore (_) to address all method signatures independent of the substituted parameter type. Finally, the defined event patterns can be aggregated and again prefixed with a label. This allows certain event patterns to be grouped so that they can be accessed in the `ORDER` section. For example, the label *Inits* aggregates two event patterns in line 24, *init(key)* and *init(key, params)*. This aggregate is then used in the `ORDER` section (line 37), simplifying the overall statement.

The **ORDER** section is integral for the whitelisting approach. With this section, it is possible to define the correct use of method calls (i.e., the correct order) by defining a regular expression over the labels and aggregates referencing method signatures from M (i.e., over the defined `EVENTS`). This regular expression induces a non-deterministic finite-state automaton $A = (Q, M, \delta, q_0, \tilde{F}) \in \mathbb{A}$ over the alphabet M, where Q is a set of states, $\delta : Q \times M \to \mathcal{P}(Q)$ the state transition function, q_0 its initial state, and finally, $\tilde{F}$ the set of accepting states. Thus, any derivation from this finite-state automaton is considered as an API misuse, including non-transition into one of the accepting states from $\tilde{F}$. Consider our example (line 37) and the equally displayed automaton in the upper part of figure 4.4. There is first the need to call

[1] https://docs.oracle.com/javase/8/docs/api/javax/crypto/Mac.html

```
SPEC javax.crypto.Mac

OBJECTS
  java.security.Key key;
  byte[] input;
  byte[] output1;
  byte[] output2;
  java.lang.String macAlgorithm;
  java.security.spec.AlgorithmParameterSpec params;
  int offset;
  int len;
  ...

FORBIDDEN
  ...

EVENTS
  g1: getInstance(macAlgorithm);
  g2: getInstance(macAlgorithm, _);
  Gets := g1 | g2;

  i1: init(key);
  i2: init(key, params);
  Inits := i1 | i2;

  u1: update(input);
  u2: update(input, offset, len);
  ...
  Updates := u1 | u2 | ... ;

  f1: output1 = doFinal(input);
  ...
  FinalsWU := f1 | ... ;
  Finals := FinalsWU | f2;

ORDER
  Gets, Inits, (FinalsWU | (Updates+, Finals))

  CONSTRAINTS
  macAlgorithm in {"HmacMD5", "HmacSHA256", ..., "PBEWithHmacSHA512"};
  offset < len;
  ...

REQUIRES
  !encrypted[output1, _];
  generatedKey[key,_];
  ...

ENSURES
  macced[output1, inp];
  ...

NEGATES
  ...
```

Figure 3.2: CrySL rule for *javax.crypto.Mac*. The rule is shortened for simplified presentation.

one of the methods aggregated by the label *Gets* (i.e., a call to *getInstance(macAlgorithm)* or *getInstance(macAlgorithm,_)*), followed by one call of *init(key)* or *init(key, params)*, which are aggregated by the label *Inits*. The next part *(FinalsWU | (Updates+, Finals))* is grouped by brackets. This means there can be either a call to one method associated by the label *FinalsWU*, which ends the correct use of the API, or (|) at least one call to a method associated by *Updates* followed by one call to *Finals*. The post-fixed (+)-operator after *Updates* indicates that it is necessary to call this method at least once, but with the possibility to call it several times in a row. In general, operators (+,?,*) usually used with regular expressions can also be used. Post-fixing a label with a question mark (?) indicates the optional call to this method. Post-fixing an asterisk (*) indicates an optional call or several calls.

The `CONSTRAINTS` section allows defining constraints on variables $v \in \mathbb{V}$ defined in section `OBJECTS`. The defined constraints C are a subset of $\mathbb{C} := (\mathbb{V} \rightarrow O \cup V) \rightarrow \mathbb{B}$. In other words, each constraint is itself a boolean function, where variables from $\mathbb{V}$ are mapped to objects in O or values with primitive types in V (more details can be found in section 3.3.3). To define such constraints, common relational operators ($<, \leq, =, \neq, >, \geq,$) and logical operators (conjunction, disjunction, implication) in combination with constants can be used (line 41). Furthermore, arithmetic operations and operations on sets are valid as well. For example, it is possible to define a constraint for a string-typed parameter that must be contained in a predefined set (line 40). Unfortunately, there are cases where such constraints are not expressive enough to describe certain usage constraints. Consider, for example, a hard-coded password wrapped in a string variable to initialize an encryption algorithm. This is particularly bad because the compiled bytecode of Java can easily be decompiled resulting the password to be leaked. To model these scenarios in terms of constraints, CrySL provides auxiliary predicates, i.e., auxiliary functions applicable to a defined variable $v \in \mathbb{V}$ from the `OBJECTS` section. In case of a hard-coded password, the predicate *notHardCoded(object)* can be used. Besides the auxialiary predicate *notHardCoded(object)*, CrySL encompasses four more such auxiliary predicates, namely *instanceof(object, type)*, *neverTypeOf(object, type)*, *callTo(method)*, and *noCallTo(method)*.

The `ENSURES` section is used to propagate the API class's proper use externally (i.e., to other rules). In other words, this section defines what the API guarantees at a specific state. Such a guarantee is defined through a predicate $p \in \mathbb{P} := \{(predName, args) \mid args \in \mathbb{V}^*\}$. Accordingly, a predicate is a pair of a predicate name and a sequence of variables. For example, in line 50 the predicate *macced[output1, inp]* is generated if the spanned automaton from the `ORDER` section is in an accepting state, all constraints are fulfilled and all predicates in the `REQUIRES` section are ensured. Expressly, by the generation of the predicate *macced[output1, inp]* the MAC (*output1*) is successfully and securely generated based on the input *inp*. Sometimes there is a need to ensure guarantees not only when the automaton is in an accepting state but *after* a specific method event pattern has been performed. Therefore, a predicate can be followed by the keyword **after** p, where p is an event pattern (i.e., referenced by a label or aggregate) defined in the `EVENTS` section. We will formally explain the generation of a predicate in the next section 3.3.3.

The `REQUIRES` section is the counterpart of the `ENSURES` section. With the definition of predicates in `REQUIRES` we can link CrySL rules to each other. In the context of cryptography, it is likely to compose several API objects properly. This means that proper use of one API object may also depend on proper prior use of another API object. In case of our example rule in figure 3.2, there are methods defined that consume a key (cf. *init(key, ...)* in line 22 and 23). The secure use of an *javax.crypto.Mac* object is also dependent on the secure and correct generation of this key. In other words, it is based on the secure and correct use of another API object (e.g., an instantiation of *javax.crypto.SecretKeySpec*). Assume there is a CrySL rule specified for *SecretKeySpec* that guarantees the proper generation of a key by exposing the predicate *generatedKey[key, inp]* under its `ENSURES` section. By defining this predicate

generatedKey[key, _] in the `REQUIRES` section (line 46) we link both rules. That is, the proper use of *javax.crypto.Mac* is only then guaranteed if the prior use of *javax.crypto.SecretKeySpec* is guaranteed as well. On the other hand, there may be predicates that are explicitly not to be ensured. For such purposes, a prefixed exclamation mark (!) to the respective predicate can be used.

In the section `NEGATES`, predicates defined in `ENSURES` that have already been ensured can be withdrawn. Assume for example the *javax.crypto.Mac* provides a method to invalidate a recently created MAC through the method *invalidate()*. Thus, with the call of *invalidate()* the guarantee on *macced[output1, inp]* (line 50) is withdrawn. Assume further, we assign the method *invalidate()* the label *invalid* in the respective CrySL rule. We could now model this scenario by specifying *macced[output1, inp] after invalid;* in the `NEGATES` section. In other words, the provided guarantee *macced[output1, inp]* is withdrawn *after* the call of the method labeled by *invalid* (i.e., the call to *invalidate()*).

3.3.3 Runtime Semantics and Formalities

With a CrySL rule, API usage constraints on the level of an API class can be defined. These usage constraints are evaluated on all runtime objects of type T specified in its `SPEC` section. The semantics of a CrySL rule is defined in terms of an evaluation on a runtime program trace. This runtime program trace contains all relevant runtime objects, values, and all events specified within the CrySL rule.

Definition 3.1 (Event [Ste20]). *"Let O be the set of all runtime objects and V the set of all primitive-typed runtime values. An event is a tuple $(m, e) \in \mathbb{E}$ of a method singature $m \in \mathbb{M}$ and an environment e (i.e., a mapping $\mathbb{V} \rightarrow O \cup V$ of the parameter variable names to concrete runtime objects and values). If the enviornment e holds a concrete object for the **this** value, then it is called the event's base object."*

Definition 3.2 (Runtime Trace [Ste20]). *"A runtime trace $\tau \in \mathbb{E}^*$ is a finite sequence of events $\tau_0 ... \tau_n$."*

Definition 3.3 (Object Trace [Ste20]). *"For any $\tau \in \mathbb{E}^*$, a subsequence $\tau_{i_1} ... \tau_{i_n}$ is called an object trace if $i_1 \leq ... \leq i_n$ and all base objects of τ_{i_j} are identical."*

For instance, the lines 13 and 14 from the example code of generating a MAC (figure 3.3) and its two events result in the following object trace:

$$(m_0, macAlgorithm \mapsto o_{algorithm}, this \mapsto o_{mac})$$
$$(m_1, macAlgorithm \mapsto o_{algorithm}, key \mapsto key, this \mapsto o_{mac})$$

Here, o_{mac} denotes that the object o is bound to the variable *mac* at runtime. $m_0, m_1 \in \mathbb{M}$ are representative for method signatures *getInstance()* and *init()* of the *javax.crypto.Mac* class. Because the static factory method *getInstance()* is used to create the respective MAC object, it is assumed that *this* is bound to o_{mac}.

To decide whether a runtime trace τ is valid according to a set of CrySL rules, the evaluation is split up into two parts. Firstly, each individual object trace is validated on its own and independently of one another. Secondly, the validation of an object trace is connected through a CrySL rule if the rule requires a guarantee from another rule (i.e., the rule consumes a predicate exposed by another rule). We briefly discuss the formalities of both steps, starting with the evaluation of individual object traces.

Individual Object Traces. The CrySL sections `FORBIDDEN`, `CONSTRAINTS` and `ORDER` are verified independently of other object traces (i.e., verified independently of other CrySL rules) but on individual object traces. Formally, an individual object trace τ^o for a runtime object o is defined as a sequence of tuples $\tau_0^o, ..., \tau_n^o$ where $\tau_i^o = (m_i^o, e_i^o)$. Here m_i^o is a concrete method call of the runtime object o and e_i^o is a concrete environment, respectively.

The `FORBIDDEN` section is validated by the function sat_F^o, which is defined as follows:

$$sat_F^o(\tau^o, F^o) \coloneqq \bigwedge_{i=0...n} m_i^o \notin F^o$$

where sat_F^o is only then satisfied if and only if no method signature m_i^o is in the set of forbidden method signatures F^o. In other words, no forbidden method is called.

The function $sat_{\mathbb{A}}^o$ checks if the given automaton induced by the `ORDER` section (i.e., the usage pattern) complies with the typestate of the runtime object o at any given time. In other words, the projection of called methods onto a sequence of method signatures $m^o \coloneqq m_0^o ... m_n^o$ from the object trace must be a word of the language $\mathcal{L}(A^o)$ where the language is induced by the automaton $A^o = (Q, \mathbb{M}, \delta, q_0, \tilde{F})$. This implies that the method signature m_n (i.e., the last called method) must lead into an accepting state of the automaton. The function sat_A^o does not consider variable bindings.

$$sat_{\mathbb{A}}^o(\tau^o, A^o) \coloneqq m^o \in \mathcal{L}(A^o)$$

Finally, every specified constraint C from the `CONSTRAINTS` section is checked for compliance. Therefore, the sequence of environments $(e_0^o, ..., e_n^o)$ of the object trace τ^o is considered. Each bounded object and values by an environment e_i^o must satisfy the constraints $c \in C^o$:

$$sat_{\mathbb{C}}^o(\tau^o, C^o) \coloneqq \bigwedge_{i=0...n} c(e_i^o)$$

Definition 3.4 (Verification of an Object Trace [Ste20])**.** *Let $\mathbb{SPEC}$ be the set of all CrySL rules. An individual object trace τ^o for a runtime object o is verified by the following function:*

$$\begin{aligned} sat^o : \mathbb{E}^* \times \mathbb{SPEC} &\rightarrow \mathbb{B} \\ [\tau^o, (T^o, F^o, A^o, C^o)] &\rightarrow sat_F^o(\tau^o, F^o) \wedge \\ &\quad\ sat_{\mathbb{A}}^o(\tau^o, A^o) \wedge \\ &\quad\ sat_{\mathbb{C}}^o(\tau^o, C^o) \end{aligned}$$

sat^o is true if and only if a given object trace τ^o satisfies its CrySL rule independently of interactions with other objects (and its respective CrySL rules), i.e., independently of the sections `REQUIRES`, `ENSURES` and `NEGATES`.

Connection of Object Traces. The `ENSURES`, `REQUIRES` and `NEGATES` sections are needed to connect individual object traces (i.e., to define interactions between CrySL rules). Formally, each state of the automaton A^o is mapped to those predicate(s) that are generated by its state. This mapping $\mathcal{G} : Q \rightarrow \mathcal{P}(\mathbb{P})$ is induced by each CrySL rule[2]. Let n^o be an event in the form of $(m^o, e^o) \in \mathbb{E}$. A predicate $p \in \mathbb{P}$ on the objects $e^o \in O$ is ensured if three (i-iii) conditions hold simultaneously.

[2]Krüger defined the mapping as $\mathcal{G} : S \rightarrow \mathcal{P}(\mathbb{P})$ but did not further defined S. However, from their work, we assume that the states of the set Q from the automaton A^o are most likely meant.

(i) An event $(m^o, e^o) \in \mathbb{E}$ leads to the state s of the automaton that ensures the generation of the predicate p (i.e., $p \in \mathcal{G}(s)$).

(ii) The function sat^o evaluates to true for the event's base object's runtime trace.

(iii) All defined predicates in the `REQUIRES` section are ensured at the time event n^o is performed and none is withdrawn.

3.3.4 Limitations

Although CrySL provides tailor-made tools to specify APIs related to cryptography, it still has limitations [Ste20]. For example, there may be *High-Level Constraints* related to temporal properties such as regular re-keying. While the JCA API does not provide any means to ensure such usage constraints, it is also not possible to specify temporal usage constraints with CrySL. In fact, CrySL is dependent on the overall functionality the API provides.

The whitelisting approach gives further limitations. Because CrySL rules are hand-crafted, and therefore, do not priori reflect the full correct usage of the API class, methods not specified in the `EVENTS` section do not affect the decision of whether the API class and its instantiation were used correctly or not.

Moreover, the application context in which an API is used is not considered. For example, the hashing algorithm *md5* is considered to be insecure when used in security contexts. Therefore, the CrySL rule specifying the correct and secure usage of the API class that provides the hashing algorithm may declare using *md5* as insecure. However, *md5* can also be used for hashing files, and indeed it often is. Thus, one cannot say in general that the use of *md5* is wrong in every case; it depends on the application context. In the case of CrySL, there is no possibility to distinguish such cases. Also, the environment in which the API is used is not taken into account. There might be cases where the correct and secure use of an API is dependent on the underlying operating system or Java version.

3.3.5 Implementation

With its clear structure and expressiveness, CrySL offers a comprehensive toolbox for defining specifications for cryptographic APIs. To define CrySL rules in a supportive way, a CrySL compiler based on Xtext[3] has been developed. Xtext offers syntax highlighting, a parser and compiler, and moreover, it is also supported by Eclipse. To be able to use CrySL in further approaches, a compiler has been developed that transforms the corresponding CrySL rule into an object model. CrySL is shipped with already defined and ready-to-use rulesets for cryptographic APIs like Java Cryptography Architecture (JCA), Bouncy Castle, and Google Tink. With these rulesets — and in combination with CogniCrypt — the authors did an in-field study on 10.000 Android apps with the result of 20.426 API misuses found [KNR+17, Ste20]. However, CrySL is a specification language, and thus, the rules recorded in this language need to be consumed by an analysis tool to actually find API misuses. Such an analysis tool is provided, among others, by CogniCrypt that we will discuss next.

3.4 CogniCrypt

CogniCrypt is an open-source tool that simplifies the often difficult application of cryptographic APIs and to detect cryptography-related API misuses [KNR+17, Ste20]. Different from other similar tools, CogniCrypt uses hand-crafted specifications recorded in the format of

[3] https://eclipse.org/Xtext/

```
1  String algorithm = "HmacSHA256";
2
3  // branch that is never taken
4  if(crossSum(15) == 0) {
5    algorithm = "NONE";
6  }
7
8  // generate key
9  KeyGenerator keygen = KeyGenerator.getInstance(algorithm);
10 SecretKey key = keygen.generateKey();
11
12 // initiate MAC
13 Mac mac = Mac.getInstance(algorithm);
14 mac.init(key);
15
16 // generate MAC
17 mac.update("Hello".getBytes("UTF-8"));
18 byte[] result = mac.doFinal();
```

Figure 3.3: An example for using *javax.crypto.Mac*[4] to generate a message authentication code (MAC).

CrySL to achieve these goals. CogniCrypt is split up into two parts: (i) CogniCrypt$_{GEN}$ generates and injects secure application of cryptography components into parts of a developers' code. (ii) CogniCrypt$_{SAST}$ is a static analysis tool to detect API misuses in code. Both parts build on top of CrySL rulesets to achieve these goals. We further refer mainly to the work of Krüger's Ph.D. thesis [Ste20].

3.4.1 CogniCrypt$_{SAST}$

CogniCrypt$_{SAST}$ can be used to perform context-sensitive and flow-sensitive static data-flow analysis on Java and Android applications. It can automatically detect API misuses based on specifications recorded in the CrySL format. However, the CrySL rules are not hardcoded. Since CrySL is a tool-independent language, the rules can also be created independently of CogniCrypt$_{SAST}$. Thus, CogniCrypt$_{SAST}$ can be considered as a static analysis tool consuming hand-crafted specifications (cf. figure 3.1). In this section we will explain how CogniCrypt$_{SAST}$ conducts the static analysis by a simplified schematic workflow 3.4, an example code 3.3 and its respective simplified CrySL rule 3.2. We further show that CogniCrypt$_{SAST}$ approximates the evaluation function sat^o (cf. section 3.3.3).

Nevertheless, before we delve deeper into how CogniCrypt$_{SAST}$ conducts the static analysis, we first consider a code example 3.3. In this example, a MAC is generated using the cryptographic API *javax.crypto.Mac*[4]. To generate a corresponding MAC, a cryptography algorithm is used, in this case *HmacSHA256* which is specified in line 1. This algorithm is used to generate the corresponding key (line 9) and for the initialization of the MAC object (line 14). In line 4, it is checked if the cross sum of 15 equals to zero. However, since the cross sum of 15 is not zero, the branch is never executed. This branch is exemplary of a non-trivial operation that can have an impact on the actual API usage. Finally, in lines 17 and 18, the MAC is generated based on the input "Hello" and stored in the byte array *result*.

CogniCrypt$_{SAST}$ is able to analyze such a piece of code displayed in figure 3.3. It first consumes all available CrySL rules, including the one for *javax.crypto.Mac*, using the CrySL compiler to parse and compile them into CrySL object models ① (see figure 3.4). With the

[4]https://docs.oracle.com/javase/8/docs/api/javax/crypto/Mac.html

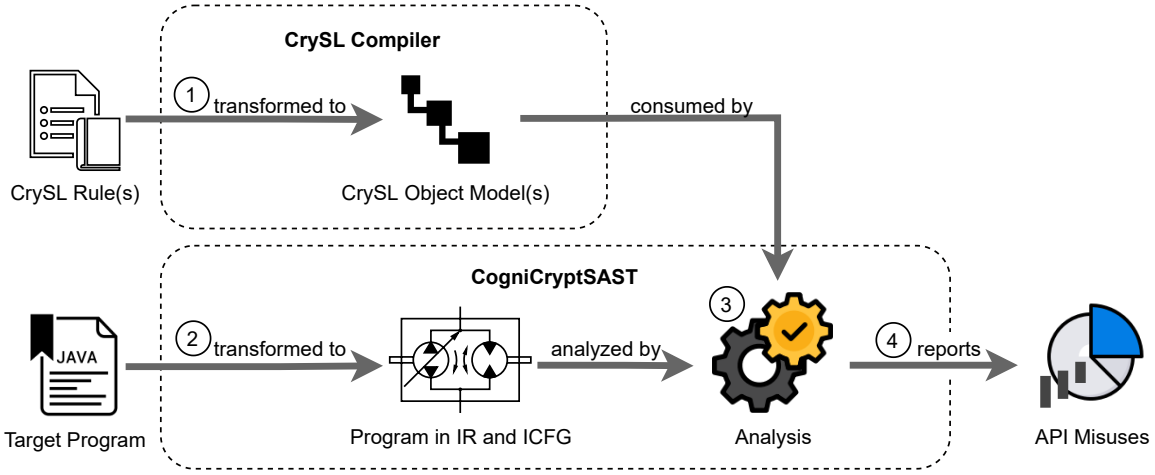

Figure 3.4: Simplified workflow of CogniCrypt$_{SAST}$. CogniCrypt$_{SAST}$ uses the CrySL compiler to transform a CrySL rule into object models, which are used, for example, to generate a state-machine. CogniCrypt$_{SAST}$ then, incorporating an IR and ICFG from the target program, performs the actual static analysis that reports any API misuses found.

help of CrySL object models, CogniCrypt$_{SAST}$ further creates artifacts used for the analysis, namely a finite-state-machine, parameter constraints, and finally, its predicates. These artifacts are representative of the `ORDER`, `CONSTRAINTS` and `ENSURES` sections. In addition, CogniCrypt$_{SAST}$ collects a list C of reference types T defined in the `SPEC` section and all their `FORBIDDEN` methods. On the other hand, CogniCrypt$_{SAST}$ uses a program analysis framework called soot [VRGH+00, LBLH11] to transform the target program's bytecode to an intermediate representation (IR) and an interprocedural control-flow graphs (ICFG) ② (cf. section 3.2).

CogniCrypt$_{SAST}$ then performs the actual analysis ③. Based on the IR and ICFG, CogniCrypt$_{SAST}$ collects all necessary information about each runtime object o that are typed by the collected list of classes C. Each runtime object o is modeled by its allocation site. Therefore, CogniCrypt$_{SAST}$ searches for *new* expressions and static calls to *getInstance()* that return an object of a relevant type to consider them as relevant allocation sites. Static methods of the form *getInstance()* are usually used in the field of cryptography as factory methods (line 9 or 13). To determine parameters and forbidden method calls, CogniCrypt$_{SAST}$ iterates through the calls along the ICFG. While CogniCrypt$_{SAST}$ iterates through the ICFG, it collects forbidden method calls. Once a forbidden method call is found, it is reported as an API misuse ④. CogniCrypt$_{SAST}$ approximates the function sat^o_F because the ICFG may also contain forbidden method calls that are never reached at runtime. However, the example code 3.3 does not contain any forbidden method calls according to the CrySL rule of *javax.crypto.Mac* API class.

CogniCrypt$_{SAST}$ then analyzes each runtime object o individually by first analyzing parameter constraints ③ (i.e., approximating the $sat^o_{\mathbb{C}}$ function). In general, two circumstances are approximated. Firstly, if constraints are not extractable from variables, for example, if the variable's value can only be determined at runtime, the constraint is considered false. Secondly, CogniCrypt$_{SAST}$ is **path-insensitive**. In this sense, CogniCrypt$_{SAST}$ considers a possible branch whether the condition is true or not. This is also the case with if-else blocks, where both blocks are considered to be executed. Therefore, in line 5, the variable *algorithm* is considered to be holding the value *NONE*, which is an invalid constraint according to the CrySL rule. However, *NONE* is no valid cryptography algorithm and an invalid constraint to the CrySL rule, and thus, reported as an API misuse ④. Nonetheless, the cross sum of 15 is not zero, and

as a result, this reported API misuse is a false positive.

The last approximation regarding sat^o is the typestate analysis on each runtime object o ③ (i.e., the validation of $sat^o_{\mathbb{A}}$). Any deviation from the state-machine is considered an API misuse and therefore reported ④. However, this typestate analysis is approximated like the analysis of constraints because branch points are considered to be executed both (i.e., both sites), which may indicate false positives.

Finally, if sat^o evaluates to true for a given runtime object o, CogniCrypt$_{SAST}$ checks that all predicates in the `REQUIRES` section are ensured ③. In case there is no specification in the `REQUIRES` section, the predicates defined under `ENSURES` are immediately ensured if and only if all previously specified constraints comply (cf. section 3.3.3). However, CogniCrypt$_{SAST}$ constantly maintains a list l with all ensured predicates, including statements responsible that a predicate is fulfilled. If the analysis requires information about predicates from other allocation sites, it consults the list l to check for compliance by matching names and parameters. For example, the generated key for creating a MAC (line 10) must be generated correctly in terms of security. Therefore, the predicate *generatedKey[key,_]* needs to be fulfilled in advance (cf. CrySL rule in figure 3.2). Thus, l is searched to ensure compliance with the requirement of a beforehand securely generated key. If such an ensured predicate is not found, an API misuse is reported ④. On the other hand, ensured predicates can be negated (i.e., killed), and thus, it is not considered to be ensured in l anymore.

CogniCrypt$_{SAST}$ is available in two versions. The first version contains the bundle of CogniCrypt$_{SAST}$ and CogniCrypt$_{GEN}$ shipped as an Eclipse plugin. This is comfortable from the developers' viewpoint as they only need to conduct the analysis to get API misuses indicated directly in their code. The second version ships CogniCrypt$_{SAST}$ as a stand-alone static analysis tool executable from the command line. By providing the target program and a CrySL ruleset, the analysis can be performed without a graphical user interface, displaying the detected API misuses by simply showing the lines of code and misuse descriptions. This allows, for example, to perform analyses in a large-scale manner.

3.4.2 CogniCrypt$_{GEN}$

Besides the analysis and detection of API misuses, there is a need for more support for developers using cryptographic APIs [NKMB16]. CogniCrypt$_{GEN}$ is an approach to bridge the gap between the complexity of using cryptographic APIs and using them correctly and securely. CogniCrypt$_{GEN}$ uses CrySL rule(s) and auxiliary templates to automatically generate correct and secure code for the usage of cryptographic APIs — presuming that the respective rule(s) define such. This automatically generated code is then injected into the developer's desired project.

CogniCrypt$_{GEN}$ is available as an Eclipse plugin and thus, provides a graphical user interface to simplify the desired code generation and injection. Since developers may have less experience in cryptography, the developer is guided in a use-case-based manner by high-level and easy-to-answer questions to elicit the users' needs. Therefore, the user needs first to select the overall task that leads to the desired goal. This can be, for example, data encryption. The user is then — among other questions — asked: "What data type do you wish to encrypt?". The developer can select *File*, *String*, *Byte Array*, or *Other/ Do not know*. With the Java file selection into which the generated code is to be injected, the code generation is completed.

However, we will not further consider the automatic code generation by CogniCrypt$_{GEN}$ as this thesis' work focuses on API misuses and its detection, respectively.

4 A Systematic Evaluation of the Expressivness of CrySL

It is yet an open question whether CrySL is expressive enough to specify correct usages of APIs outside the context of cryptography. This open question also includes how well CogniCrypt$_{SAST}$ can detect API misuses in code. So far, no effort has been made to evaluate CrySL and CogniCrypt$_{SAST}$ to API misuses related to non-cryptographic APIs [Ste20]. In this chapter, we present our conducted manual evaluation of CrySL and CogniCrypt$_{SAST}$ to uncover the strengths and weaknesses of both based on our classification framework of API usage constraint types 2.4.1. Our results will show that CrySL and CogniCrypt$_{SAST}$ are generally not applicable to APIs outside the domain of cryptography without further adaptation.

We first present the conducted datasets as well as the overall evaluation process in section 4.1. The results of our evaluation, including the weaknesses and strengths of CrySL and CogniCrypt are presented in section 4.2. However, our evaluation may not be representative of all possible API misuses in all domains to which APIs may be tailored. Thus, we discuss in section 4.3 the threats of validity to our results. In section 4.4, we provide reasons why CrySL offers insufficient language constructs when applied to non-cryptographic APIs by comparing the application of cryptographic APIs to non-cryptographic APIs. We summarize the results of this chapter in section 4.5, where we also provide answers to **RQ3** and **RQ4** based on our observations.

4.1 Evaluation

To conduct a systematic evaluation, we first introduce two datasets (section 4.1.1). We take these datasets as a basis for the systematic evaluation process explained in section 4.1.2. We then present the results of our evaluation in section 4.1.3.

4.1.1 Considered Datasets

To conduct the evaluation, we use API misuses from the *MUBench Dataset* [ANN+16] and the most frequent API misuse pattern found by Li [Li20]. The *MUBench Dataset* comprises API misuses mainly from bug-datasets like *BugClassify* [HJZ13] or *Defect4J* [JJE14]. Most of the API misuses are related to real projects and can also be tracked via version-control systems. All API misuses in the *MUBench Dataset* are pre-classified by *MuC* [ANN+19] (cf. section 2.5). For each API misuse, the *MUBench Dataset* provides a YAML-file [BKEI09], which contains basic information such as a description about the API misuse, the commit URL from the respective version-control system, and the classified API misuse type. Moreover, for each API misuse,

a pattern is provided that fixes the actual API misuse. Because we want to focus on non-cryptographic API misuses, we excluded cryptographic API misuses. Besides, Li introduced the five most common API misuse patterns for each of their API misuse categories (cf. section 2.5), which appeared in bug-fixing commits from GitHub as part of their large-scale empirical study on API misuses.

MUBench Dataset. To obtain statistical information about the *MUBench Dataset* [ANN+16] and the respective API misuses, the authors provided a Docker container[1]. With this Docker container, basic statistical queries on the dataset are possible, such as querying how many API misuses it contains. However, for our purpose, this is insufficient because of two reasons. Firstly, we want to systematically evaluate API misuses by their API and the category in which they appear. Secondly, an initial sample of API misuses showed several conceptually same API misuses exist, but only in different projects. We do not want to evaluate the same conceptual API misuse several times because it does not add any value to the evaluation. Therefore, we have developed a Python program that shows an overview of all APIs and their API misuses, and the absolute number of occurrences and sorted under the categories of *MuC*. Additionally, it allows filtering for specific APIs and their API misuses paired with the API misuse description. Since the description for equal API misuses remains the same, we can easily skip conceptually equal API misuses. In total, we reviewed all 99 different API misuses from the *MUBench Dataset*, of which 51 are related to the *Java Class Library*[2], 17 to *Apache APIs*[3], 11 to the *Android API*[4], and 20 to miscellaneous APIs. 23 of all API misuses we reviewed were no API misuses by our definition (cf. section 2.3).

Five most frequent API misuse patterns. Li collected API misuses in bug-fixing commits from open-source projects on GitHub [Li20] (cf. section 2.5). Unfortunately, their dataset is not yet publicly available at the time of this writing. However, they mention the five most common API misuse patterns from each category they introduced in their classification framework. For example, the method *equals(String)* from the *java.lang.String* API class is the most common misuse pattern concerning their *Missing Exception* category. Their category *Missing Exception* aims at missing exception handlers added with the respective commit to wrap the API usage. In total, we reviewed all 43 different of their introduced misuse patterns. 38 misuse patterns were related to the *Java Class Library*, one to *Apache APIs* and 4 to miscellaneous APIs. Nevertheless, these five most commonly encountered misuse patterns for each API misuse category should be viewed critically with doubt regarding an API misuse by our definition (cf. definition 2.4). This is because the authors define an API misuse as "[...] code edit operations related to some APIs in a bug fix". Thus, the mentioned API misuse patterns do not necessarily have to be counted as API misuse according to our definition since an API usage can also be changed without being due to an API misuse in a bug-fixing commit. This assumption is confirmed with a look at their category *Redundant Exception*. This *Redundant Exception* category aims at API usages within a try-catch block which API usage is then removed within the bug-fixing commit. One of the top five frequent API misuse patterns in *Redundant Exception* relates to the usage of the *java.lang.Exception*[5] API class and its method *getMessage()*. We believe this method is not removed from a try-catch block because it is called at an improper place; it is rather removed because *getMessage()* is often used for debugging purposes and thus, simply removed. However,

[1] https://hub.docker.com/r/svamann/mubench
[2] https://docs.oracle.com/javase/8/docs/api/allclasses-frame.html
[3] https://apache.org/
[4] https://developer.android.com/reference
[5] https://docs.oracle.com/javase/8/docs/api/java/lang/Exception.html

we reviewed the top five most frequent API misuse patterns for each category. Unfortunately, this review is only approximate since Li only mentioned the API misuse pattern (i.e., the fully qualified API class name and the corresponding method call). This means we lack contextual information that is often needed to identify the actual API misuse. Therefore, we were only able to consider 12 of the 43 API misuse patterns as API misuses according to our definition.

4.1.2 Evaluation Process

To standardize and record the results of the evaluation, we created an evaluation protocol template with mainly four sections: (i) Overall information about the API, (ii) the misuse pattern, (iii) the correct pattern, and finally, (iv) identified problems with CrySL and CogniCrypt$_{SAST}$ as well as first improvement suggestions. Such a protocol is created for each API misuse. An example protocol can be seen in appendix 8.1.

The evaluation process for an API misuse is shown in figure 4.1. For each alleged API misuse, we first created a protocol based on the evaluation protocol template ①. Each protocol consists of at least the specified fully-qualified API class name and the misuse pattern ②. If official documentation was available online for the respective API class, then we have added the link accordingly. To make sure we address an API misuse by our definition 2.4, we considered the respective API specification to identify API usage constraints that may have been specified ③. If we determined that the alleged API misuse is none, we justified that decision in the protocol ④. This also includes API misuses mentioned by Li that could not be evaluated due to missing contextual information. If the alleged API misuse complies with our definition, we tried to specify the API usage constraint in a CrySL rule to cover the API misuse scenario ⑤. We stopped evaluating that particular API misuse and recorded the problems accordingly in case of non-specifiable API usage constraints ⑧. If the correct API usage constraint was specifiable, we implemented the API misuse in a test project. Since the *MUBench Dataset* contained API misuses, preferably from real-world projects, it was necessary to abstract them from their original context to their essentials to focus on the actual API misuse ⑥. We then conducted the static analysis by running CogniCrypt$_{SAST}$ on the test project that contains the API misuse ⑦. Finally, we recorded the evaluated problems as well as first improvement suggestions ⑧. Whenever we found misbehavior or bugs related to the CrySL compiler or CogniCrypt$_{SAST}$, we created an issue tracker on GitHub[6].

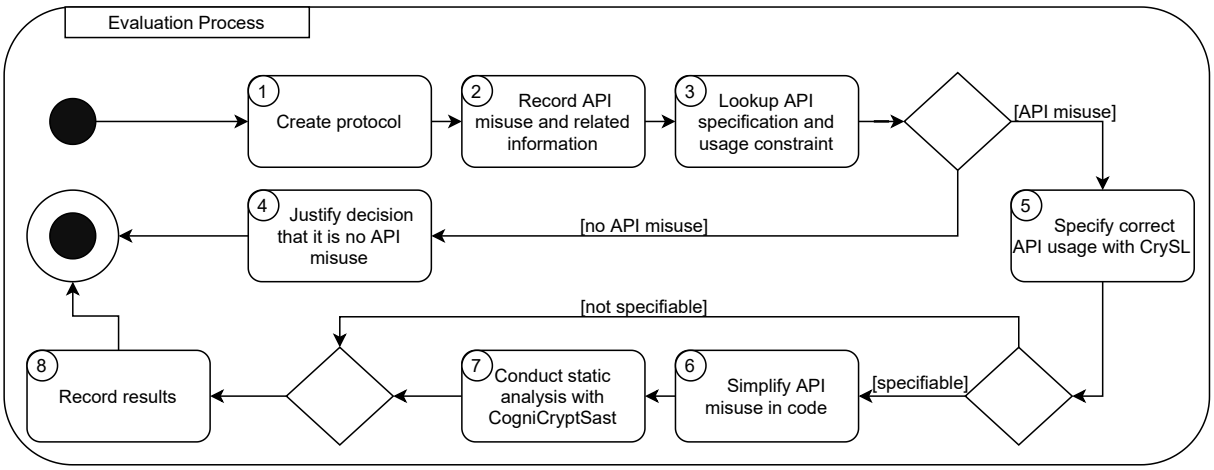

Figure 4.1: Activity diagram representing the overall evaluation process.

[6] https://github.com/CROSSINGTUD/CryptoAnalysis/issues

4.1.3 Results

The results of the evaluation are presented in table 4.1. The boldly highlighted rows are the parts of an API method call and summarize the results from each usage constraint type linked to it. Unfortunately, no API misuses are classifiable as violations of *Post-Call(s)*, *Argument Type*, *Argument Correlation*, and *High-Level Constraints* (grey colored rows). Therefore, we can only give a theoretical answer about the expressiveness of CrySL related to these API usage constraint types.

The first column (from left) lists the API usage constraint types from our classification framework (cf. section 2.4.1). The next column *TC* (total count) provides the total number of classified API misuses (i.e., respective violations of the API usage constraints) for each category. Out of 142 alleged API misuses in total, 88 meet our definition of an API misuse.

The overlapping column (*O*) shows the number of API misuses that have already been classified in another category but are additionally classified in this category (cf. section 2.4.3). For example, consider an API that requires checking null on a return value of a method call classified as *Post-Null-Check* constraint. However, in addition to check the return value for

Table 4.1: Results of manual assessment. Grey-colored rows indicate no API misuses were assessed in the respective category. green/orange/red colored cell ≙ in summary feasible/perhaps feasible[6]/not feasible, *TC* ≙ Total Count, *O* ≙ Overlapping, *S/PS/NS* ≙ specifiable/perhaps specifiable[6]/not specifiable, *D/PD/ND* ≙ detectable/perhaps detectable[6]/not detectable, *F/PF/NF* ≙ feasible/perhaps feasible[6]/not feasible

API Usage Constraint	**Misuses**		**CrySL**			**CogniCrypt$_{SAST}$**			**Feasibility**		
	TC	O	S	PS	NS	D	PD	ND	F	PF	NF
Receiver	**16**	**0**	**0**	**0**	**16**	**0**	**0**	**16**	**0**	**0**	**16**
Post-Call(s)											
Post-Null-Check	16	0	0	0	16	0	0	16	0	0	16
Method Call	**42**	**1**	**19**	**3**	**21**	**5**	**17**	**21**	**5**	**17**	**21**
Method Call Sequence	19	0	19	0	0	5	14	0	5	14	0
Controlling Method Call	20	1	0	0	21	0	0	21	0	0	21
Forbidden Method Call	3	0	0	3	0	0	3	0	0	3	0
Passed Arguments	**10**	**6**	**5**	**0**	**11**	**3**	**2**	**11**	**3**	**2**	**11**
Argument State	1	0	1	0	0	0	1	0	0	1	0
String Format	3	6	3	0	6	3	0	6	3	0	6
Number Range	1	0	1	0	0	0	1	0	0	1	0
Argument Type											
Argument Correlation											
Pre-Null-Check	5	0	0	0	5	0	0	5	0	0	5
Multiple Assignments	**20**	**36**	**0**	**0**	**56**	**0**	**0**	**56**	**0**	**0**	**56**
Exception Handling	14	36	0	0	50	0	0	50	0	0	50
Context	1	0	0	0	1	0	0	1	0	0	1
Synchronization	4	0	0	0	4	0	0	4	0	0	4
Threading	1	0	0	0	1	0	0	1	0	0	1
High-Level Constraints											
Summary	***88***	***43***	***24***	***3***	***104***	***8***	***19***	***104***	***8***	***19***	***104***

[6]With perhaps, we mean that it may be specifiable or detectable when the related bug is fixed.

null in an "if"-statement, there is also the possibility of wrapping the further use of the return value in a try-catch block. Thus, it is also possible to comply with the API usage constraint using proper exception handling by reacting to a potentially thrown exception, in this case to *NullPointerException*. As a result, all reviewed API misuses classified under the API usage constraint *Post-Null-Check* are also classified as overlapping under *Exception Handling*. Another example are API misuses classified as overlapping under the API usage constraint *String Format*. Consider, for example, regular expressions that are usually recorded as string literals. Now there may be APIs like *java.lang.String* that provide methods that consume regular expressions. Such methods require only passing regular expressions in a valid format. However, to respond to incorrect formatted regular expressions, the only opportunity is to catch a possible thrown exception. Therefore, we classified such cases under *Exception Handling* and *String Format* as overlapping, respectively.

To be able to make a statement about problems regarding CrySL as well as CogniCrypt, we evaluated both separately from each other (cf. section 4.1.2). However, suppose an API usage constraint is not specifiable with CrySL. In that case, the respective API misuse is also declared as not detectable under CogniCrypt$_{SAST}$ and consequently, not feasible in the last column *NF*. API usage constraints that could be specified in CrySL but the corresponding API misuse was not detected by CogniCrypt$_{SAST}$ were also specified as not feasible (*NF*).

Because we encountered bugs regarding the CrySL compiler and CogniCrypt$_{SAST}$, we were not able in these cases to evaluate them. Nevertheless, API misuses in these cases are likely to be perhaps feasible (*PF*) (i.e., specifiable and detectable if the respective bug gets fixed). Therefore, we counted them to the columns perhaps specifiable (*PS*) in case of bugs related to the CrySL compiler as well as perhaps detectable (PD) in case of CogniCrypt$_{SAST}$. We will introduce the encountered bugs in section 4.2.

Our evaluation's overall results are displayed in the colored cells of table 4.1 in the lower right-hand corner. The results include API misuses that were counted multiple times due to overlap. Overall, we could only specify the API usage constraint and detect the corresponding API misuse in 8 cases (green cell). Most of the successfully identified API misuses are related to API usage constraints of category *Method Call Sequence*. These API misuses are related to violations of the actual order in which methods must be called (cf. section 2.4.1.2). We can observe that these API misuses are similar to those found in the context of cryptography — the API usage is targeted, often tied to a specific goal, and ultimately has a defined beginning and end of use. For example, the API class *java.io.PrintWriter*[7] allows writing character sequences to a resource (e.g., a file). Thus, the overall goal that can be achieved using *PrintWriter* is to write character sequences to the resource until the stream is closed with a call to *close()*. These targeted API usages can be specified with CrySL using the `ORDER` section. Furthermore, CogniCrypt can detect a deviation of the specified method call order (i.e., the API misuse). CrySL offers another precise tool with the specification of constraints on passed string literals that must consist of a predefined set. Violations of such constraints are detected by CogniCrypt$_{SAST}$ in any case. For example, the *java.lang.String* API class provides a method to convert a string literal to a byte array by calling *getBytes(String charsetName)*. With CrySL we can restrict the passed argument *charsetName* to a predefined set of standardized and supported charsets[8], such as UTF-8. However, this may lead to false positives as the developer may want to use a character set that is not listed in the predefined set in the corresponding CrySL rule but is allowed to be used by the system that supports the character set. Thus, the restriction on string literals is only approximate. On the other hand, CrySL also provides expressiveness to cover API usage constraints from type *Number Range*. Although we only analyzed one API misuse related to

[7] https://docs.oracle.com/javase/8/docs/api/java/io/PrintWriter.html
[8] https://docs.oracle.com/javase/8/docs/api/java/nio/charset/Charset.html

Number Range that could be successfully specified, we argue that it is feasible to specify similar constraints restricting the range of passed numbers.

1. Observation: Strengths of CrySL

CrySL is expressive in cases where the API usage is targeted and associated with a very specific goal, and ultimately has a defined beginning and end of use. In addition, CrySL provides enough expressiveness to cover most API usage constraints related to *Method Call Sequence*, *Forbidden Method Call*, *Argument State*, *String Format* and *Number Range*.[9]

Unfortunately, we could not specify API usage constraints or detect the respective API misuses in most cases. Even if we exclude all API misuses classified as overlapping (43 in total) that are not feasible and count those that could be feasible if an associated bug is fixed (19 in total) to feasible ones, the total number of unspecifiable or not detectable API misuses is still high (61 out of 88 API misuses, 69%).

4.2 Evaluated Problems

CrySL provides essential expressiveness for API usage constraints such as *Method Call Sequence*. Nevertheless, some API usage constraints lack possibilities to specify them. Furthermore, even if an API usage constraint is specifiable with CrySL, CogniCrypt$_{SAST}$ encountered problems to detect and indicate the respective API misuse in code. In the following section 4.2.1 we use our classification framework of API usage constraints (cf. section 2.4.1) to discuss problems and respective scenarios we encountered during our evaluation process. Furthermore, in section 4.2.2, we want to point out conceptual problems related to inheritance as the concept of inheritance is assigned much greater importance using APIs outside the context of cryptography. This, in turn, is also reflected in the use of CrySL rules.

4.2.1 Problems related to API Usage Constraints

We first discuss the weaknesses and flaws of CrySL and CogniCrypt$_{SAST}$ that we encountered during our evaluation process for each category of our classification framework.

Receiver. We reviewed 16 API misuses in total related to the receiver of a method call. All 16 API misuses are related to API usage constraints of *Post-Null-Check* and none are specifiable or detectable. Unfortunately, the reviewed datasets (cf. section 4.1.1) do not contain any API misuses related to *Post-Call(s)*. However, it is theoretically not possible to specify the required method calls on the receiver for three reasons. Firstly, the specified method call sequence in the `ODER` section is due to CrySL's runtime semantic evaluated on each runtime object individually (cf. section 3.3.3) whereas it is assumed that an object trace τ^o either starts with a *new* expression or through static factory methods like *getInstance()* exposed by the API. Thus, the specified method call sequence is not evaluated on the receiver created by other methods. Secondly, the receiver of a method call can only be bound to variables defined in the `OBJECTS` section, for example, for further defining a predicate, but not to define API usage constraints in the form of required method calls on it. Thirdly, the receiver does not necessarily have to be of the same type as the API. The only interaction between different API classes is to provide

[9] Taking into account that not specifiable API usage constraints due to bugs may be fixed in the future.

guarantees by defining a predicate in the `ENSURES` section. Nevertheless, requirements to call methods on a receiver is not a guarantee but a restriction on further use, and therefore predicates cannot be used to specify required post calls. In the case of *Post-Null-Check*, CrySL provides no facilities to specify such API usage constraints.

2. Observation: Weaknesses of CrySL

CrySL does not provide expressiveness to specify *Post-Call(s)* and *Post-Null-Check* usage constraints. In general, it is not possible to specify API usage constraints to address restrictions or requirements on called methods related to the receiver of a method call.

Method Call. We reviewed 42 API misuses related to the method call (and one overlapping), from which 19 are specifiable and 5 API misuses are detectable. In all 19 cases, we could specify a *Method Call Sequence* in the `ORDER` section that can be used to detect the respective API misuse. Although it seems *Method Call Sequence* constraints are very well specifiable with CrySL, we argue that the whitelisting approach will end in problems in practice. This is because we specified the method call sequence with the particular API misuse in mind. In other words, we defined the method call sequence in such a way that the particular API misuse could be detected. In practice, however, this may lead to problems. Consider for example the code snippet displayed in figure 4.2.

The *javax.swing.JFrame*[10] API can be used to create graphical frames. Whenever the frame is created, and components are added, the frame needs to be adapted to its content before it is made visible to the user. Therefore, *pack()* needs to be called before *setVisible(true)*. Furthermore, we consider not calling *pack()* before *setVisible(true)* as an API misuse. We could now declare the regular expression *(p, s)* where p is the label for *pack()* and s for *setVisible(true)* respectively. Yet, this is insufficient in practice because whenever there is a call to p, then the call to s is expected, which will lead to possible false positives. This is because there may be other correct usages after the call to p that are not directly followed with the call to s. Moreover, there is no API usage constraint that enforces the call to s after p and thus, calling p does not require calling s at all. Overall, the only way to specify a regular expression that will hold in practice is to model all possible correct usages beginning with the call to p until calling s. This includes all valid sequences between p and s and also all valid sequences that do not include the call to s. We argue that such scenarios will heavily blow up the regular expression and end in illegibility of the rule. Another problem is related to non-specific APIs. Likely, they do not enforce any method call sequence (cf. definition 2.1), which causes the `ORDER` section to be empty. However, this is not allowed by CrySL as it expects to specify at least one method that needs to be called.

```
1 JFrame f = new JFrame("Hello World");
2 // ... add components
3 f.pack();
4 f.setVisible(true);
```

Figure 4.2: An example of a subsequence. The *javax.swing.JFrame* API provides functionality to create a frame. To display the frame properly, *pack()* need to be called before *setVisible(true)*.

[10] https://docs.oracle.com/javase/8/docs/api/javax/swing/JFrame.html

Not only CrySL but also CogniCrypt has problems figuring out the method call sequence in code related to exception handling. For example, method calls in a finally block, which are always executed, are not considered. Furthermore, CogniCrypt does not consider method calls that are chained and sometimes indicates false positives. More details can be consulted on the respective issue trackers[11][12][13] on GitHub.

Furthermore, we reviewed 20 (one overlapping) API misuses related to *Controlling Method Call* constraint. Unfortunately, CrySL does not provide any language constructs to specify *Controlling Method Call* constraints. In the case of *Forbidden Method Call*, CrySL allows the specification of forbidden methods. However, all three forbidden method calls we considered expect a string to be passed. Due to a bug[14] we were not able to specify all of them.

3. Observation : Weaknesses of CrySL

The whitelisting approach becomes a problem in patterns where one method forces another method to be called before, resulting in a bloated and illegible regular expression. In addition, there are API classes that do not impose any constraints on the method call order but other constraints, whereas CrySL expects that every API class imposes a method call sequence. In the case of *Controlling Method Call* constraints, no language constructs are provided to specify them.

Passed Arguments. We reviewed 10 API misuses (and six overlappings) related to passed arguments of a method call from which five are specifiable, and three related API misuses are detectable. One API usage constraint related to *Argument State* could be specified but not evaluated because of a bug[15] regarding CogniCrypt$_{SAST}$ where the generated predicates in the `REQUIRES` section are not correctly recognized when used in a rule that consumes them. API usage constraints of the type *String Format* are specifiable and detectable in every case where a passed string literal needs to be in a predefined finite set of values (three cases in total). In the remaining six cases, the passed string literal needs to be a valid string from an infinite set of values — for example, a valid regular expression. Thus, such API usage constraints are not specifiable with CrySL as no infinite sets are specifiable under `CONSTRAINTS` section. However, in all six non-specifiable cases, we noticed that an exception is thrown when an invalid string literal is passed. Therefore, it is overlapping with the API usage constraint of type *Exception Handling*. In the case of *Number Range*, the API usage constraint could successfully be specified (i.e., only passing numbers greater or equals than zero to *java.lang.Thread.sleep(number)*[16]). However, the related API misuse is not detected by CogniCrypt$_{SAST}$ due to a bug[17] as the Thread-API-class is not considered correctly.

We could not evaluate the categories *Argument Type* as well as *Argument Correlation* as the considered datasets do not contain any API misuses accordingly. However, it is theoretically possible to specify API usage constraints of type *Argument Type* since CrySL provides the auxiliary predicate *instanceof(object, type)* (cf. section 3.3) which allows for restricting the type of the passed argument. In the case of *Argument Correlation*, we can model dependencies between passed primitive values. For example, it is possible to constraint a method *example(Integer a,*

[11] https://github.com/CROSSINGTUD/CryptoAnalysis/issues/315
[12] https://github.com/CROSSINGTUD/CryptoAnalysis/issues/313
[13] https://github.com/CROSSINGTUD/CryptoAnalysis/issues/334
[14] https://github.com/CROSSINGTUD/CryptoAnalysis/issues/317
[15] https://github.com/CROSSINGTUD/CryptoAnalysis/issues/318
[16] https://docs.oracle.com/javase/8/docs/api/java/lang/Thread.html#sleep-long-
[17] https://github.com/CROSSINGTUD/CryptoAnalysis/issues/314

Integer b) with $a \geq b$ in the `CONSTRAINT` section. However, we cannot give a more precise answer due to the lack of API misuses.

Finally, none of the *Pre-Null-Check* constraints are specifiable because no language constructs are provided by CrySL to define them.

4. Observation: Weaknesses of CrySL

With CrySL we cannot specify *String Format* constraints that restrict the passed string literal to be in an infinite set of valid values. Furthermore, no language constructs are provided to specify *Pre-Null-Check* constraints.

Multiple Assignments. In total, we reviewed 20 API misuses related to API usage constraints (and 36 overlapping) that had multiple assignments, namely, *Exception Handling*, *Context* and its subcategories *Synchronization* and *Threading*. Unfortunately, none of them are specifiable with CrySL.

In the case of the *Exception Handling* constraint, CrySL does not provide any facilities to specify the proper exception handling. Thus, we were not able to specify nor detect any 50 reviewed API misuses. The specification of proper exception handlers is not only crucial because of API misuses related to the exception type that is caught, for example, when a too general exception type like *Throwable* is caught (cf. API usage constraint 2.4.1.4). The specification of *Exception Handling* constraints is important for two more reasons.

Firstly, we observed that in many cases, exception handling could be used as a backup to comply with other API usage constraints such as *String Format*. This is especially the case when there is no possibility to check for compliance of API usage constraints before using the API and the value of passed arguments is only known at runtime. Consider for example the *java.util.regex.Pattern*[18] API class that provides the method *compile(regex)*, which requires only passing regular expressions in the form of a string. Thus, *compile()* enforces the API usage constraint *String Format* by only passing strings that are actually valid regular expressions. However, the API class does not provide functionality to check a passed regular expression to be valid, but only to catch for *PatternSyntaxExcpetion*. Thus, the only way to comply with *String Format* is to catch the respective exception.

Secondly, the proper exception handling influences required method calls (i.e., the *Method Call Sequence* constraint). Consider for example the *java.io.FileReader*[19] API class again. The method call sequence is specified as follows: Whenever the API's object is in a state that allows reading a single character file, *read()* can be called arbitrary often. In the end of usage there is the requirement to release the resource by calling *close()* (see also state-machine in figure 2.3). In figure 4.3, a possibly thrown *IOException* is caught, but if *read()* throws the exception, the

```
1 FileReader r = new FileReader("file"); // file exists
2 try {
3   r.read(); // may throw exception
4   r.close(); // not executed in case of a thrown exception
5 } catch(IOException) { /** recover **/  }
```

Figure 4.3: The code snippet shows a possible violation of *Method Call Sequence* constraint if *read()* throws an exception which results in *close()* not being executed.

[18] https://docs.oracle.com/javase/8/docs/api/java/util/regex/Pattern.html
[19] https://docs.oracle.com/javase/8/docs/api/java/io/FileReader.html#FileReader-java.io.File-

method call sequence is truncated and thus no longer conforms to the API usage constraint. Such scenarios are not considered by CogniCrypt$_{SAST}$ nor is it possible to specify the correct exception handling in CrySL.

5. Observation: Weaknesses of CrySL

CrySL does not provide expression facilities for specifying restrictions related to *Exception Handling* constraints. However, specifying the correct exception handler is essential not only because of missing or incorrectly set exception handlers but also because it can be used as a backup for compliance with other API usage constraint types. Moreover, proper exception handling is also crucial for detecting API misuses regarding the method call sequence that may be interrupted by a potentially thrown exception.

In total, 6 API misuses are classifiable to *Context* constraint and its subcategories *Synchronization* and *Threading*. One API misuse is classified as *Context*, four as *Synchronization* and another one as *Threading* while none of the respective constraints are specifiable with CrySL. In case of the API misuse related to *Context* constraint, the API misuse aims at not calling *wait()* exposed by *java.lang.Object*[20] in a loop. In the case of *Synchronization* constraint, all four API misuses do not reveal any similar patterns. One API misuse causes a deadlock because of obtaining the lock twice in a nested manner. The second API misuse results from not obtaining a lock because the API object is structurally modified in a multi-threaded environment. The third API misuse refers to an API object that is iterated over in an unsynchronized manner, resulting in non-deterministic behavior. The fourth API misuse lacks synchronization on the return value of an API object's method. Thus, all investigated *Synchronization* constraints tackle different aspects of an API usage in multi-threaded environments. In the case of the subcategory *Threading*, we reviewed only one API misuse that aims at not using *javax.swing.JFrame* components on the required event dispatching thread [Orab].

6. Observation: Weaknesses of CrySL

Neither *Context*, *Synchronization*, nor *Threading* constraints are specifiable with CrySL due to lack of provided language constructs.

Unfortunately, no API misuses and corresponding API usage constraints can be categorized as *High-Level Constraints*. However, as Krüger [Ste20] already mentioned, CrySL is limited to time-related constraints such as regular re-keying (also see section 3.3.4). Because we lack examples, we can neither give a statement to this category nor give a statement to CrySL or CogniCrypt.

4.2.2 Inheritance

During our evaluation, we encountered open questions not related to API usage constraints but to the programming concepts of inheritance — i.e., the programming paradigms of specialization and generalization. Specialization is the paradigm to implement more refined and specific concepts by deriving from a less specific class. Conversely, the generalization paradigm states that specialized classes derived from a less specific class have similarities in their concept [Tai96].

[20] https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html

Considering CrySL, no language constructs are provided to represent inheritance structures in a CrySL rule. In general, the rule is specified on the API depending on its fully-qualified class name. The conducted analysis of CogniCrypt$_{SAST}$ considers only runtime objects typed by the specified reference type in the CrySL rule. In other words, whenever we create CrySL rules — for example, for an interface or an abstract class — they are only then considered by CogniCrypt$_{SAST}$ when the receiver of the factory method or constructor is typed by the reference type specified in `SPEC` section. This means that specified API usage constraints on less specific classes need to be repeated in any derived class considering the specialization paradigm. For example, assume we declare an API usage constraint for the *java.util.Iterator* API class that requires checking for elements in the iterator by calling *hasNext()* before receiving any element by calling *next()*. Now, the API designer needs to declare exactly this API usage constraint again in all subinterfaces and implementing classes to align to the programming paradigm. Conversely, whenever the developer takes advantage of the generalization paradigm, the CrySL rule of the less specific class is used, which is correct. However, the API designer/expert needs to take care of the type-hierarchy and make sure that there are also CrySL rules for superclasses.

In summary, the only possibility to align to specialization and generalization is to manually take care of all enforced API usage constraints to align to these programming principles.

4.3 Threats to Validity

We examined API misuses from the *MUBench-Dataset* [ANN+16] and the five most frequent API misuse patterns from Li [Li20]. Nevertheless, the API misuses we considered may not represent all possible API misuses in the wild. Li have already noted that the API misuses from the *MUBench-Dataset* may not be sufficient to evaluate the capabilities of static analysis tools. It takes a significantly larger number of different API misuses to provide a meaningful result in terms of a static analysis tool's capabilities. Furthermore, the majority of the API misuses we considered (62%) are related to APIs from the Java Class Library (JCL). However, as pointed out in the previous section 4.4, domain-specific usage constraints and thus, API misuses may exist not only in the domain of cryptography but also in other domains such as machine learning. Therefore, we believe that more domain-specific API usage constraints and API misuses will emerge with the consideration of other domain-specific APIs.

4.4 The Difference to Cryptography related Constraints and Misuses

CrySL has its difficulties with respect to specifying API usage constraints outside the context of cryptography (cf. section 4.2). Nevertheless, this does not mean that CrySL is overall a bad specification language. In this section, we explain why CrySL's expressiveness is insufficient when applied to non-cryptography-related constraints. Therefore, we take a look at the differences between using cryptographic and non-cryptographic APIs.

If we consider the domain of cryptography, APIs in this area are widely used to safeguard sensitive data in order to prevent eavesdropping or forgery [Ste20]. Cryptographic APIs implement conceptually secure algorithms. In Java, these algorithms are then offered to the outside world through APIs - for example, the *Java Cryptography Architecture* (JCA)[21]. Besides choosing the right API class for a particular task, it is necessary to use its instantiated objects as they are intended to be used. In other words, the correct use of a cryptographic API object is predefined, has a very beginning and end of use, and only very few correct usages. The

[21] https://docs.oracle.com/javase/8/docs/technotes/guides/security/crypto/CryptoSpec.html

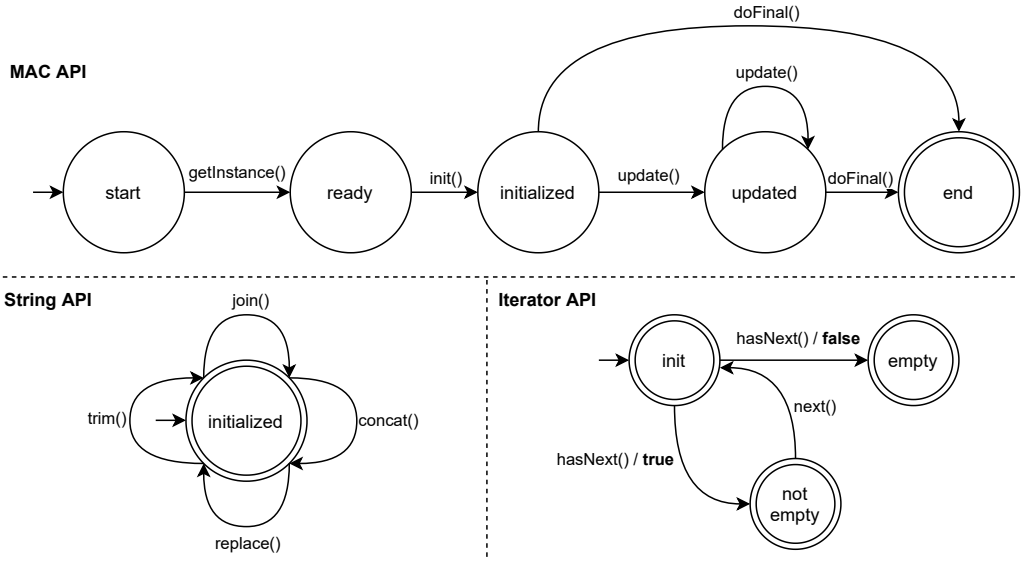

Figure 4.4: Simplified finite-state-machines of *javax.crypto.Mac*[22] (upper), *java.lang.String*[25] (lower left-hand side) and *java.util.Iterator*[24] (lower right-hand side). The instantiated MAC object is properly used with a transition into the final state *end* and no derivation from its underlying state-machine. In contrast, the String-API class enforces no sequences to be followed. In case of the Iterator, only one optionally subsequence is present (i.e., calling *next()* requires to be in *not empty* state).

proper use of cryptographic APIs is crucial for ensuring security properties, as many studies show [NKMB16, LCWZ14, EBFK13, Ste20, WRE+19].

4.4.1 Method Calls

We can observe that cryptographic API objects are (i) used in a very goal-driven way and (ii) have few proper uses but many ways of being misused [Ste20]. This is because cryptographic APIs follow predefined theoretical protocols very closely. Consider, for example, figure 4.4. A simplified finite-state-machine that defines the correct use of a *javax.crypto.Mac*[22] object is displayed in the upper part. The *javax.crypto.Mac* API class provides functionality to generate a message authentication code (MAC) based on a message that can be used to check the integrity. The MAC object is only correctly used when the methods are invoked in no other order than the specified method call sequence. Thus, the usage of a MAC object starts with the factory method *getInstance* and ends with a transition into the final state *end*. In other words, the goal of the MAC object is achieved by generating the desired MAC. Any deviation from the specified sequence(s) from the *start* state to the *end* state, including non-transition to the *end* state, is considered as an API misuse. For example, calling *doFinal* while the MAC object is in state *ready* is considered as an API misuse. In general, we can likely describe the whole correct use of a cryptographic API object with a sequence of possible method calls: a cryptographic API enforces a *Method Call Sequence* constraint over its full use. We can also observe that cryptographic objects are only needed until the desired goal is achieved (e.g., generating the MAC). Thus, cryptographic objects have a very beginning and end of use. This peculiarity is reflected in the concept of CrySL, as the `ORDER` section reflects exactly this API usage constraint type. This is also the reason why conceptual similar API classes like the *java.io.PrintWriter*[23]

[22] https://docs.oracle.com/javase/8/docs/api/javax/crypto/Mac.html
[23] https://docs.oracle.com/javase/8/docs/api/java/io/PrintWriter.html

can be easily specified (cf. section 4.1.3).

Consider now the simplified finite-state-machine of a *java.lang.String*[25] in the lower left-hand part of figure 4.4. We can see that there is no given sequence because there is only one final state (*initialized*) and all edges lead back to that state. This means that the API object can be used appropriately in a wide context without restrictions on method calls in very different ways. Moreover, this means that the API class *String* does not specify an inherently predefined goal that the developer must achieve in a certain prescribed way. This is different from cryptographic APIs because they are used to achieve a particular goal requiring the correct invocation of methods in a specific order.

The finite-state-machine of *java.util.Iterator* shown in the lower right-hand part is similar to *java.lang.String*. Every state is a final state, which means that there is no need to transition into a final state that defines the end of use. On the other hand, this also shows that the API does not specify a certain goal described by reaching a final state. Rather, it depends on the developer what exact goal he is pursuing with the use of the API object. The only difference to the String-API is that there is a controlling method call. Whenever *next()* is called, the API object requires to be in *not empty* state. This means that we need to safeguard the call to *next()* by wrapping it in a branch that is executed depending on the iterators' state. Only when the iterator reveals the state *not empty*, the method call *next()* is allowed. In addition, we observed that non-cryptographic APIs rather span smaller subsequences where a method call requires a previous different method call (cf. section 4.2). For example, the *javax.swing.JFrame* API class requires calling *pack()* before *setVisible(true)* to adapt the frame to its content. This is different from cryptographic APIs because we can specify their objects' correct use throughout their lifetime. In addition, we cannot observe constraints of type *Controlling Method Call* in the context of cryptography, and thus, CrySL provides no language constructs to specify them.

Furthermore, we noticed differences regarding the *Argument State* usage constraint type (cf. section 2.4.1). When using cryptographic APIs, there is often a need to compose several API objects in a particular way to achieve a certain goal [Ste20]. Thus, it is very likely to connect method call sequences between different API objects. However, we have noted that API usage constraints of type *Argument State* are not prevalent to such an extent in non-cryptographic APIs.

4.4.2 Strings, Integers and Exception Handling

We can observe that byte arrays, integers, and strings are often used to parameterize specific cryptographic algorithms [Ste20, WRE+19]. The proper parameterization is crucial for the security of the used cryptographic APIs. For example, an integer can be used to define the iterations performed by a hash function. However, the given integer influences a hash function's security: if the given number of iterations does not meet a given threshold, it is considered insecure. Thus, the chosen integer can lead to vulnerability. This is also the case if we consider strings. These are often used to specify the cryptographic algorithm used. Specifying an incorrect string argument can lead to vulnerabilities if an insecure cryptographic algorithm is used. Overall, these API usage constraints can be modeled in CrySL's `CONSTRAINTS` section by requiring a passed argument to be in a set of predefined values or to be greater than a given threshold.

In contrast, predefined allowed values occur much less in non-cryptographic APIs as they are enforcing API usage constraints on string or integer values depending on the API objects' state. For example, integers are often used as keys to refer to elements in a list or array. To comply with such usage constraints, controlling method calls or exception handling are used. In fact,

[24] https://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html
[25] https://docs.oracle.com/javase/8/docs/api/java/lang/String.html

exceptions have only a minor role in cryptographic APIs, since only in a few cases an exception is thrown when the API is used insecurely [Ste20]. This can also be seen in the expressiveness of CrySL, since there are no possibilities to include exception handling in the specification.

4.4.3 Summary

Based on our observations, we generally argue that the more generic the use case(s) of an API is/are, the fewer restrictions exist on allowed, required, and forbidden method calls. Moreover, APIs outside the context of cryptography can have various use cases; for example, if we consider the *java.lang.String* API class. We observed that non-cryptographic APIs are not used in a specific way with a very beginning and end of use but rather span multiple smaller subsequences and controlling method calls that depend on an API object's state, each fulfilling a different goal. In addition, subsequences are not required by the API to be traversed; they are often optional in nature and are only considered required if the subgoal of the API wants to be achieved by the developer. We can say that the use of cryptographic APIs differs significantly from that outside of cryptography and thus, also their API usage constraints. This is the reason why CrySL often does not provide facilities to cover API usage constraints outside the domain of cryptography.

4.5 Conclusion

In this chapter, we presented our systematic and manual evaluation of CrySL and CogniCrypt$_{SAST}$ to uncover strengths and weaknesses of both based on our classification framework of API usage constraint types (cf. section 2.4.1). We tried to specify API usage constraints in CrySL rules to uncover a total of 142 different non-cryptography-related API misuses collected by two studies [ANN+16, Li20]. We found that 23 API misuses are actually none by our definition (cf. section 2.3). 31 API misuses could not be further evaluated because too little contextual information was available. Unfortunately, from 88 API misuses left, we were only in 8 cases successfully able to specify the API usage constraints in CrySL rules and to detect the corresponding API misuses. These specifiable constraints and detectable misuses were related to violations of *Method Call Sequence* and *String Format* usage constraints (cf. section 2.4.1). However, because of bugs we found — and once they are fixed — it is likely that 19 additional API usage constraints could be specified and subsequently, their violations could be found without any adaption of CrySL and CogniCrypt$_{SAST}$. We have shown that CrySL is limited in its provided language constructs to specify API usage constraints outside the context of cryptography. There are no language constructs provided to specify *Exception Handling*, *Controlling Method Call*, *Pre-Null-Check* and *Post-Null-Check* as well as *Context*, *Synchronization* and *Threading*. Furthermore, we have also shown that exception handling plays an important role outside the context of cryptography. The proper exception handling influences the method call sequence and can also be used to comply with other API usage constraints such as *Post-Null-Check*. All in all, this answers **RQ3** as CrySL and CogniCrypt$_{SAST}$ do not apply to APIs outside the context of cryptography without further improvements. This also implies that we are not able to tackle **RQ4** because evaluating precision and recall requires applicability to non-cryptographic APIs. However, our results also show that CrySL is overall a domain-specific language and perfectly tailored to the needs of its domain, cryptography. This is the main reason why CrySL provides only language constructs tailored to the need of its domain but no other. Since non-specific APIs or other domain-specific APIs enforce other API usage constraints, CrySL is not expressive enough to cover them. Furthermore, this raises the question of how we can enrich CrySL in its expressiveness to cover the non-specifiable API usage constraints. We tackle this approach in the next chapter.

5 Proposals for increasing the Expressiveness of CrySL

CrySL lacks language facilities to specify certain types of API usage constraints, for example, *Exception Handling* or *Controlling Method Call* (cf. section 4.2). In this chapter, we propose additional language facilities to improve CrySL (**RQ5**). In section 5.1, we justify to which API usage constraint type we propose improvements. We then introduce language facilities to enrich the expressiveness of CrySL. In section 5.3, we summarize our proposals and their disadvantages.

5.1 Focus on particular API Usage Constraints

The API usage constraint types of our classification framework (cf. section 2.4.1) sometimes appear with significantly different prevalence in API documentation. Also, their prevalence differs in the respective violation (i.e., in the occurrence of API misuses — cf. section 2.6). We will therefore focus on improvement approaches based on the prevalence of each constraint type.

Table 5.1: Simplified results of the evaluation of CrySL and CogniCrypt$_{SAST}$ (cf. section 4.1.3) for each API usage constraint type. (Filled) circles symbolize the feasibility of specifying API usage constraint types and respectively detecting their violations, i.e., API misuses. ○ ≙ no API usage constraints were specifiable and its violations not detectable respectively, ◐ ≙ at least one API usage constraint was specifiable and its violation detectable, ● ≙ every API usage constraints were specifiable and its violations detectable[1].

Receiver	
Post-Call(s)	○
Post-Null-Check	○

Passed Arguments	
Argument State	●
String Format	◐
Number Range	●
Argument Correlation	●
Argument Type	●
Pre-Null-Check	○

Method Call	
Method Call Sequence	◐
Controlling Method Call	○
Forbidden Method Call	●

Multiple Assignments	
Exception Handling	○
Context	○
Synchronization	○
Threading	○
High-Level Constraints	○

API usage constraints assigned to **Method Call** are widespread in API documentation (up to 16.4%), and those are also the overall most misused ones (up to 70.2% in bug datasets and 54.0% in the wild). If we consider table 5.1, only *Controlling Method Call* constraints are not specifiable[1]. Although we could specify all *Method Call Sequence* constraints, we encountered the problem of truncating a method call sequence because of a possible thrown exception. Furthermore, specifying the proper exception handling will also serve as a backup for other constraints, such as *Pre-Null-Check* or *String Format* (cf. section 4.2.1). Thus, we will first focus on proposing improvements for *Controlling Method Call* and *Exception Handling*.

API usage constraints related to **Passed Arguments** are the most frequent in API documentation (22.2%) but less misused than those related to a method call (56.8% in bug datasets and 15.0% in the wild). Only *Pre-Null-Check* constraints are not specifiable. In the case of *String Format* we were only in those cases not able to specify the usage constraint where the passed value needs to be from an infinite set of valid values. However, those will also be covered by the proposal for the proper exception handling (cf. section 4.2.1). Thus, we focus on proposing improvements for specifying required null checks, which also includes *Post-Null-Check* constraints.

The API usage constraints listed under **Multiple Assignments** occur quite frequently in API documentation (up to 16.4%) but are violated somewhat less (13.6% in bug datasets and 2.3% in the wild). In the case of *High-Level Constraints*, we lack data to provide a fundamental proposal. Furthermore, in the case of *Threading* and *Context*, we only evaluated one API usage constraint each. In the case of *Synchronization*, we evaluated four API usage constraints which were all different without common patterns (cf. section 4.1). While we could suggest language facilities specifying these constraints in CrySL, the better approach is to elaborate these particular types first to make then a fundamental approach (cf. chapter 6).

The remaining constraints linked to the **Receiver** contains of *Post-Null-Check*, which will be addressed alongside *Pre-Null-Check*, and *Post-Call(s)*. Unfortunately, we did not evaluate any API misuses related to *Post-Call(s)* because of lacking data. Moreover, we do not have data of its violations' prevalence, and also, it is the least common usage constraint type in API documentation. Therefore, we will not make any proposals for this usage constraint type and refer to chapter 6.

5.2 Suggestions for Improvements

In this section, we provide proposals for improvements with a focus on particular API usage constraint types (cf. previous section 5.1), i.e., *Exception Handling*, *Controlling Method Calls*, *Post-Null-Check* and *Pre-Null-Check*. As an example, we present the syntax of the proposals for *Exception Handling* in the form of a context-free grammar (Extended Backus-Naur Form [BBG+63]). We also formally introduce runtime semantics for *Exception Handling* following Krüger's [Ste20] scheme (cf. section 3.3.3). However, we explain our approach for the remaining proposals without going into syntax elements explained by grammar or runtime semantic formalities and leave it to future work to introduce them. Our overarching goal is to adhere to CrySL's design rules of simplicity and tool-independent semantics (cf. section 3.3.1). All suggested language facilities are theoretical and not yet implemented in CrySL.

5.2.1 Exception Handling

To specify the proper exception handling we propose a new section `EXCEPTIONS` (see line 6 in figure 5.1). The API designer/expert can specify all relevant exceptions possibly thrown by

[1] Assuming that bugs related to CrySL and CogniCrypt$_{SAST}$ are fixed in the future.

```
1  SPEC java.io.FileReader
2
3  OBJECTS
4    java.io.File f;
5
6  EXCEPTIONS
7    fnf: FileNotFoundException;
8    ioe: IOException;
9    agr:= fnf | ioe;
10
11 EVENTS
12   i: FileReader(f) throws fnf;
13   r: read() throws ioe;
14   c: close() throws ioe;
15
16 ORDER
17   i, r*, c
```

Figure 5.1: Simplified CrySL rule for *java.io.FileReader*[2]. For example, *read() throws ioe* indicates that *read()* possibly throws *IOException* at runtime.

```
1 FileReader r
2   = new FileReader("file");
3 try{
4   r.read(); // throws IOException
5   r.close(); // API misuse
6 } catch(IOException e) {
7   // recover
8 }
```

Figure 5.2: An API misuse of *java.io.FileReader*[2]. The misuse occurs because *read()* throws an *IOException* resulting into not calling *close()*. However, *close()* is required to be called at the end of *FileReader*'s usage to release the resource.

a method in the `EXCEPTIONS` section. All exceptions are specified with a label. Furthermore, exceptions can be aggregated and assigned to a new label (line 9). The labels can be used to specify the possible thrown exception by a method in the `EVENTS` section after the keyword *throws* (lines 12-14).

Considering figure 5.2, an API misuse is shown in line 5. We assume that *read()* throws an *IOException*. As a result, the required method call sequence specified in the `ORDER` section is truncated because *close()* is not executed. By specifying the possible exceptions thrown by methods in the respective CrySL rule, the analysis tool (e.g., CogniCrypt$_{SAST}$) could detect such scenarios and indicate a possible API misuse to the developer.

Furthermore, by specifying possible thrown exceptions, we can also tackle two more scenarios. Firstly, the analysis tool can check for exceptions that are possibly wrong (e.g., *Throwable* is caught instead of the specified one, which might suppress other exceptions) or are not caught at all. Secondly, the specified exceptions can back up other API usage constraint types such as *String Format* (cf. section 4.2).

However, in the case of a static analysis tool, we cannot say with certainty that an API misuse is present in all the scenarios addressed. This is because a static analysis tool suffers the required runtime context that is needed to report a violation of *Exception Handling* constraints with certainty. We can observe a violation of *Exception Handling* constraints if and only if the exception is thrown at runtime and the exception is not caught properly. For example, in the case of the API misuse scenario shown in figure 5.2, the case may occur that *read()* does not throw an *IOException* and the required method call sequence holds (i.e., there is no API misuse). Thus, without the required runtime context, a static analysis tool like CogniCrypt$_{SAST}$ can only indicate some kind of warning in these cases.

The relevant syntax elements are shown as an excerpt of the grammar in figure 5.3 and 5.4. The first figure shows all relevant syntax elements needed to specify elements for exception handling. An exception is specified by its label and its class name (*EXCLABEL : exception-classname*). Already specified exceptions can be aggregated by their labels (*EXCLABEL :=*

[2] https://docs.oracle.com/javase/8/docs/api/java/io/FileReader.html

EXCLABEL ::=
label

EXCAGGR ::=
EXCLABEL | EXCAGGR
EXCLABEL

EXCEPTION ::=
EXCLABEL := EXCAGGR
EXCLABEL : exceptclassname — *A: B — an exception class name B labeled by A*

PARAMETERS ::=
varname , PARAMETERS
varname

METHOD ::=
methname(PARAMETERS)

EVENTAGGR ::=
label | EVENTAGGR
label

EVENT ::=
label := EVENTAGGR
label : METHOD

EVTHROWS ::=
EVENT
EVENT throws EXCLABEL — *A throws B — an event A throws exception labled by B*

Figure 5.3: Excerpt of relevant CrySL syntax elements for exception specification proposal in Extended Backus-Naur Form (EBNF) [BBG+63].

EXCEPTIONS
EXCEPTIONS ::=
EXCEPTION ; EXCEPTIONS ; — A ; B — a list of exceptions A and B
EXCEPTION ; — A — a list of the single exception A

EVENTS
EVENTS ::=
EVTHROWS ; EVENTS ; — X ; Y — a list of events X and Y
EVTHROWS ; — X — a list of the single event X

Figure 5.4: Excerpt of the CrySL rule syntax in Extended Backus-Naur Form (EBNF) [BBG+63] displaying relevant syntax elements for exception specification proposal.

EXCAGGR). The second figure 5.4 shows the two CrySL sections `EXCEPTIONS` and `EVENTS` highlighted in bold letters. However, the specification of an exception after an event (i.e., a specified method) is optional, as the production rule *EVENTS* is a list of one or more events, whereas an event can be specified with an exception or not (*EVTHROWS*).

The runtime semantics of the proposal is defined as follows. Each CrySL rule is validated along each runtime object o specified by its reference type T from the `SPEC` section (cf. section 3.3.3). We extend the definition of an event $(m, e) \in \mathbb{E}$ that contains not only the method signature $m \in \mathbb{M}$ and the environment $e : \mathbb{V} \rightarrow O \cup V$ (i.e., a mapping of the variables to concrete runtime objects and values) but also a mapping $p : \mathbb{P} \rightarrow P$. Here, $\mathbb{P}$ is the set of all specified exceptions in the `EXCEPTIONS` section mapped to P (i.e., concrete caught exceptions in code surrounding the respective method call of m). We assume the mapping considers the type-hierarchy of exception classes. Each element in P is unique in its identifier with respect to the try-catch-finally construct in which it is caught.

Let $\tau_1^o ... \tau_n^o$ be an object trace, where $\tau_i^o = (m^o, e^o, p^o)$ and $1 \leq i \leq n$ (cf. section 3.3.3). Let further $X_i \subseteq \mathbb{P}$ all possible thrown exceptions by the method signature m_i specified in the CrySL rule. We consider two cases in which a warning is issued.

In the first case, a warning is issued if the mapping p_i^o of one of the events $\tau_1^o ... \tau_n^o$ does not contain a caught exception, i.e., $x \notin p_i^o(x)$ for $x \in X_i$. In other words, the called method m_i^o of the runtime object o is not surrounded by a try-catch block that catches one of the specified exception $x \in X_i$, possibly causing the program to crash.

The second case deals with the scenario where a method call sequence is truncated because two required method calls are called in the same try-catch block. Here, the first method call of both throws an exception that is caught in the catch block, but the second method call is not called in either the catch or the finally block. Before we explain the formalities for this case, we need to restrict its application model. We adopt a simplified model by accepting the called method as executed in terms of the required method call sequence, even if the called method throws an exception. In other words, if the method call sequence *(a, b)* is required but *a* throws an exception, we assume *a* is still executed properly in terms of the method call sequence. The formalities are specified as follows: The `ORDER` section reveals a method call sequence $m_1 ... m_n$. Let τ_i^o be the concrete event of m_i and τ_k^o the concrete event of m_k with $i < k \leq n$. Three (i-iii) conditions need to hold to indicate a warning:

(i) τ_i and τ_k are found in the same try-block.

(ii) $\exists x \in p_i^o(x)$ for $x \in X_i$ (i.e., a specified exception x of the method m_i is caught in the catch block).

(iii) There is no concrete event τ_l^o with $k < l \leq n$ for the possibly not executed method m_k whereas τ_l^o is expected in the following catch or finally-block.

In other words, (i) two method calls of a required method call sequence are found in the same try block, (ii) whereas the possible thrown exception of the first method call is caught. (iii) Because the second method call is not called in the catch or finally block (i.e., called in case of a thrown exception of the first method call), the method call sequence is truncated in this case. Thus, a warning is issued.

We have introduced the syntax elements and formalities regarding *Exception Handling* constraints based on the reviewed API misuse examples during our evaluation. However, there may be corner cases that are not covered. Therefore, we refer to chapter 6, to first examine the different API usage constraint types and their violations in-depth to uncover possible existing corner cases.

5.2.2 Controlling Method Call

To be able to specify API usage constraints of type *Controlling Method Call*, we propose another new section called `GUARDS` (line 6 in figure 5.5). A guard safeguards another method call that may only be called if the API object's state allows the invocation. Consider, for example, figure 5.6. Assume an empty iterator object is passed to the method *firstElem(Iterator it)*. Calling *next()* on an empty iterator resulting immediately into a thrown *NoSuchElementException* and is accordingly an API misuse causing the program to crash. We can ensure that the thrown exception does not cause a program to crash in two ways. Firstly, *NoSuchElementException* is caught (cf. line 2-5 in figure 5.7). Secondly, the call to *next()* is safeguarded by *hasNext()* in a control structure, for example, in an "if"-statement (cf. line 9 in figure 5.7). In this case, *hasNext()* reveals the API object's state. Only if *hasNext()* returns true (i.e., the API object's state allows to call *next()*), we can say with certainty that *next()* will not lead to any error and will return the desired entry from the iterator. In figure 5.5, the respective guard is specified in line 7. The guard requires calling *hasNext()* in a control structure and its return value need to be checked for true. The label of the guard is then used to be specified for the method *next()* after the keyword *guarded by*. Because *next()* possibly throws *NoSuchElementException* (line 10), the guard *negates* such possibly thrown exception (line 7). This means, whenever the guard surrounds the respective method call, there is no need to catch the exception that is negated by the guard.

Guards can also be checked for other constants than boolean values (e.g., numeric constants). Furthermore, all classical logical operators ($<$, $\leq$, $=$, $\neq$, $\geq$, $>$) can be used. Besides constants, there may also be the need to check the API object's state to a passed argument. Consider, for example, figure 5.9. Retrieving an element from an *ArrayList*[4] with an index that is greater than the array list's size results into a thrown *IndexOutOfBoundsException*. Thus, the guard (line 10

```
1  SPEC java.util.Iterator
2
3  EXCEPTIONS
4    n: NoSuchElementException;
5
6  GUARDS
7    h: hasNext() == true negates n;
8
9  EVENTS
10   n: next() throws n guarded by h;
```

Figure 5.5: Simplified CrySL rule for *java.util.Iterator*[3]. *next() throws n* indicates that *next()* may throw *NoSuchElementException*. However, if *next()* is *guarded by n* (i.e., surrounded by a control structure with *hasNext() == true*), then *NoSuchElementException* never will be thrown (i.e., *throws n* gets negated).

```
1  // assume empty iterator is passed
2  public void firstElem(Iterator it) {
3    it.next();
4  }
```

Figure 5.6: An API misuse of *java.util.Iterator*[3]. Calling *next()* on an empty iterator results into a thrown *NoSuchElementException*.

```
1  public void firstElemV1(Iterator it) {
2    try {
3      it.next();
4    } catch(NoSuchElementException e) {
5      /** recover **/
6    }
7  }
8
9  public void firstElemV2(Iterator it) {
10   if(it.hasNext()) it.next();
11 }
```

Figure 5.7: Appropriate calls to *next()* in case of an empty passed iterator.

[3] https://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html
[4] https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html

```
1  SPEC java.util.ArrayList
2
3  OBJECTS
4    int index;
5
6  EXCEPTIONS
7    n: IndexOutOfBoundsException;
8
9  GUARDS
10   l: size() > index negates n;
11
12 EVENTS
13   i: get(index) throws n
14     guarded by l;
```

Figure 5.8: Simplified CrySL rule for *java.util.ArrayList*[4]. The guard *l* requires to check *size()* greater than the specified object *index* which is used in the event *get(index)*.

```
1 // assume index > a.size()
2 public void acc(ArrayList a, int index) {
3   a.get(index);
4 }
```

Figure 5.9: An API misuse of *java.util.ArrayList*[4] resulting the program to crash.

```
1 public void acc(ArrayList a, int index) {
2   if(a.size() > index) a.get(index);
3 }
```

Figure 5.10: Appropriate call to *get(index)* as it is properly safeguarded.

in figure 5.8) requires to check the *size()* of the array list to be smaller than the passed argument *index* which will negate a possible thrown *IndexOutOfBoundsException*. An appropriate call to *get(index)*, which satisfies the guard of the CrySL rule, is shown in figure 5.10. Finally, the method specified in the guard does not necessarily need to be a method exposed by the API class but from other API classes as well. Consider again the CrySL rule shown in figure 5.1. The constructor *FileReader(f)* throws a *FileNotFoundException* if the *java.io.File*[5] object (line 4) passed points to a file *f* that has not been created yet (line 12). We could now specify the guard *l: file.exists() negates fnf;* and adapt line 12 to *i: FileReader(f) throws fnf guarded by l;*. Thus, specifying a guard with a method exposed by another API class is also possible.

However, a static analysis cannot decide whether a true or false positive in case of a *Controlling Method Call* violation is present because a missing guard does not necessarily mean the usage constraint is violated. For example, if the passed *ArrayList* in figure 5.9 contained at least as many elements as the passed index, there would be no API misuse and no exception handler or guard is needed. Thus, the static analysis tool can only indicate some kind of warning.

5.2.3 Null Checks

The third introduced improvement tackles API usage constraint types of *Pre-Null-Check* and *Post-Null-Check*. However, it is unnecessary to propose a new section, but the red highlighted keywords that are shown in figure 5.11. Consider, for example, figure 5.12. The constructor *File(path)* requires to not pass a null value. Otherwise, a *NullPointException* is thrown. In the case of a passed null-string, the *Pre-Null-Check* constraint is violated in line 2 (i.e., an API misuse is present). These *Pre-Null-Check* constraints can be specified as shown in line 10 of figure 5.11. With *p => not null throws n* we specify a possible thrown *NullPointerException* declared in line 7. Either the passed string is checked for null in advance, which negates the possible thrown *NullPointerException* — just like in the `GUARDS` section —, or the possible exception is caught by proper exception handling. Because the API designer/expert may influence the thrown exception type, it is possible to specify the exception type thrown in case of a violation. However, the static analysis can only indicate a violation of a *Pre-Null-Check* constraint if the

[5] https://docs.oracle.com/javase/8/docs/api/java/io/File.html

```
1  SPEC java.io.File
2
3  OBJECTS
4    java.lang.String p;
5
6  EXCEPTIONS
7    n: NullPointerException;
8
9  EVENTS
10   i: File(p => not null throws n);
11   g: getParent() returns null;
```

Figure 5.11: Simplified CrySL rule for *java.io.File*[5]. The constructor *File(p)* requires passing only values that are not null. When null is passed, the *File(p)* will throw *NullPointerException*. *getParent()* properly returns null. Thus, further actions on the return value requires checking for null.

```
1 public void parent(String path) {
2   File f = new File(path);
3   String p = f.getParent();
4   p.getBytes();
5 }
```

Figure 5.12: Code snippet shows two possible API misuses. In case *path* is null, *Pre-Null-Check* constraint is violated in line 2. Otherwise, if a valid path is passed, but the path has no parent, the string *p* is null (line 3). Thus, the call to *getBytes()* throws an *NullPointerException* and is accordingly a violation of the *Post-Null-Check* constraint.

null value is extractable in code without execution. Otherwise, only a missing null check or a missing exception handler can be indicated (i.e., some kind of warning).

In the case the passed string *path* is a valid path to a directory. However, the directory does not have any parent and thus, a violation of *Post-Null-Check* constraint is present in line 4. In this case, *getParent()* returns a null value. Calling a method on a null-object immediately results in a thrown *NullPointerException*. We can specify a possible returned null value with the keywords *returns null* after the specified method in the `EVENTS` section (line 11 in figure 5.11). Unlike specifying *Pre-Null-Check* constraints, we do not need to specify the type of exception thrown because accessing or modifying the field of a null object, or calling methods on a null value immediately results in a thrown *NullPointerException* [Oraa]. Also, in the case of *Post-Null-Check*, a missing check for null does not necessarily mean the usage constraint is violated as there may be cases where no null value is returned. Thus, in the case of detecting violations of *Post-Null-Check* constraints, the static analysis can only indicate some kind of warning.

5.3 Conclusion

In summary, the enhancements presented in this chapter answer **RQ5**. With just a few minor improvements in CrySL and two new sections, namely `EXCEPTIONS` and `GUARDS`, we would be able to specify API usage constraints for four additional API usage constraint types, namely, *Post-Null-Check*, *Pre-Null-Check*, *Exception Handling* and *Controlling Method Call* (see table 5.2). With an appropriate integration in CrySL and CogniCrypt$_{SAST}$, we estimate that theoretically, 82 out of 88 of the API misuses we reviewed can be found (92% in total). This is an increase of three times more reported API misuses compared to the current capabilities of CrySL and CogniCrypt$_{SAST}$. In case of *Post-Call(s)*, *Context*, *Synchronization*, *Threading* and *High-Level Constraints*, we are lacking API usage constraints and API misuse examples to propose a fundamental approach to tackle these types.

Our proposed language facilities may be suitable for indicating our reviewed API misuses. However, we believe an integration will result in many unnecessary warnings being displayed, which will heavily inflate the API misuse report. As a result, this would tremendously weaken the applicability of CogniCrypt$_{SAST}$. This is because, for example, a missing exception handler

for an unchecked exception is not directly an API misuse. Only when an exception is thrown at runtime which is not caught properly, the *Exception Handling* constraint is violated (i.e., an API misuse is present). Only in the case of violations of *Pre-Null-Check* constraints, an API misuse can be indicated when the passed variable contains a null value that is extractable from code. For all other proposed extensions, only a warning can be displayed since static analysis inherently lacks runtime context and can therefore not decide in these cases whether a true or false positive is present. To validate the displayed warnings of the static analysis, we propose an approach in the next chapter.

Table 5.2: Simplified results of the evaluation of CrySL and CogniCrypt$_{SAST}$ (cf. section 4.1.3) contrasted with proposals introduced in this chapter. **Bef.** shows feasibility of CrySL and CogniCrypt$_{SAST}$ based on our evaluation. **Aft.** shows feasibility if proposals would be integrated. Dots symbolize the feasibility of specifying API usage constraint types and respectively detecting their violations, i.e., API misuses. ○ ≙ no API usage constraints are specifiable and its violations not detectable respectively, ◐ ≙ at least one API usage constraint is specifiable and its violation detectable, ● ≙ every API usage constraints are specifiable and its violations detectable.

Receiver	**Bef.**	**Aft.**
Post-Call(s)	○	○
Post-Null-Check	○	●

Passed Arguments	**Bef.**	**Aft.**
Argument State	●	●
String Format	◐	●
Number Range	●	●
Argument Correlation	●	●
Argument Type	●	●
Pre-Null-Check	○	●

Method Call	**Bef.**	**Aft.**
Method Call Sequence	◐	●
Controlling Method Call	○	●
Forbidden Method Call	●	●

Multiple Assignments	**Bef.**	**Aft.**
Exception Handling	○	●
Context	○	○
Synchronization	○	○
Threading	○	○
High-Level Constraints	○	○

6 Outlook

In this chapter, we present four topics for ongoing work that could follow from this thesis. The first topic can be seen independently of CogniCrypt$_{SAST}$ and CrySL. However, it would help to improve API misuse detection and specification languages of API usage constraints in general, whereas the remaining topics are related to improvements of CogniCrypt$_{SAST}$ and CrySL. All the proposed ideas are sorted in descending order of importance from our point of view.

More detailed exploration of API usage constraint types. Our evaluation has shown that two types of knowledge are required to determine the capabilities of API misuse detection tools (e.g., CogniCrypt$_{SAST}$) and specification languages of API usage constraints (e.g., CrySL): A comprehensive database of API misuse and API usage constraint examples, and a comprehensive and precise classification framework of its types.

In the case of API misuse detectors, its capabilities can only be meaningfully determined once a high-quality database with sufficient different (and validated) examples of API usage constraints and its violations (i.e., API misuses) are available. While the *MUBench Database* [ANN+16], for example, provides some examples of API misuses and their correct applications, it has already been noted in the work of Li [Li20] that its examples may be insufficient to meaningfully capture the capabilities of API misuse detectors. This is confirmed as there is a lack of data examples for violations of *Post-Call(s)*, *Argument Type*, *Argument Correlation*, and *High-Level Constraints* types (cf. table 4.1). Furthermore, during our evaluation, we found that some examples present in *MUBench Database* do not violate API usage constraints and are therefore no API misuses (cf. section 4.1). Also, Li's approach of providing a more comprehensive set of API misuse examples comes with its problems. For example, we argue that the technical analysis of API misuses based on bug-fixing commit messages does not necessarily yield actual API misuses, which is reflected in the top five most frequently appeared fixing patterns for each of their API misuse categories. For instance, the most often removed method in their mined bug-fixing commits related to their API misuse category *Exception Handling* is *Exception.getMessage()*[1] (cf. section 2.5). We believe that removing the method call is not due to API misuses but is often used because of debugging purposes, and therefore simply removed. In other words, the API misuse examples in their dataset are not validated to be actual API misuses.

On the other hand, to evaluate (and develop) specification languages such as CrySL, it is necessary to fully understand the domain for which it is being developed and applied. In particular, if a specification language is being developed for specifying ubiquitous API usage

[1] https://docs.oracle.com/javase/8/docs/api/java/lang/Exception.html

constraints (i.e., not for a specific domain like that of cryptography), the different types of API usage constraints need to be known. Although our classification framework (cf. section 2.4) summarizes the API usage constraint types from a wide variety of previous work, we discovered evidence during our evaluation that additional subtypes exist, for example, in the case of *Synchronization* constrain type (cf. section 4.2.1). However, we were not able to confirm this due to a lack of data.

To provide the most comprehensive database of API misuse examples, API usage constraints, and the most comprehensive classification framework to date, we propose a possible — and extensive — experimental setup of a large-scale study shown in figure 6.1. Firstly, there is a need to mine a comprehensive set of code examples using (specific) APIs ①, e.g., from GitHub[2]. These code examples can be used to mine common API usage patterns ② [NHO18, NNP+09]. In a second step, alleged API misuse examples can be used from the dataset mined by Li [Li20] containing over 950.000 bug-fixing commits ③. Since similar alleged API misuse examples are likely in Li's dataset (cf. section 4.1.1), they are grouped to similar alleged API misuse examples ④. While common API usage patterns can be used to identify API misuses, however, this assumption is based on the fact that a deviation from common API usage patterns constitutes an API misuse [Ama18] (cf. section 3.1). To support the validation of actual API misuses and constraints, the API documentation ⑤ can be used to mine API usage constraints through a natural-language processing approach ⑥ [LLS+18, LLY+20, RYX+20]. With an appropriate matching procedure, API misuse examples, common API usage patterns, and API usage constraints can be merged into a set of triples ⑦. This allows a direct comparison of them. If the API usage constraint was successfully mined from documentation, it could be used to validate the API misuse example automatically ⑧. However, API documentation may be inadequate or missing (cf. section 2.3), resulting in a possibly not extractable API usage constraint. This means that manual validation is the only option left ⑧. However, the possibility of direct comparison could make this process significantly more performant. Once a triple is validated, API usage constraint types can be derived to enrich our classification framework (cf. section 2.4) or *MuC* [ANN+19] ⑨. To take advantage of the benchmarking functionality of *MUBench* [ANN+16], it can be enriched with the validated data ⑩.

With such a comprehensive database and classification framework, scientists would be able to significantly improve their specification languages and API detection tools. On the one hand,

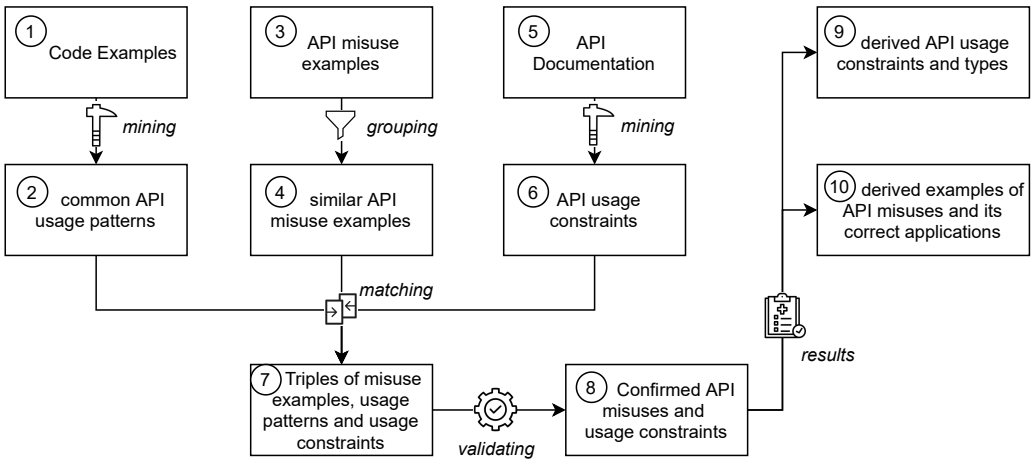

Figure 6.1: Possible experimental setup of a large-scale study to identify validated API misuses and API usage constraints.

[2] https://github.com/

it allows highlighting possible weaknesses of their approaches (e.g., not covered detection of violations of certain API usage constraint types). On the other hand, it allows the comparison of different approaches by measuring precision and recall, both values being meaningful due to the extensive data. Furthermore, the correct application examples can also be used to improve API misuse fixing approaches [NHKO20]. Note that the experimental setup shown needs to be further investigated and checked for feasibility for its implementation.

Implementation of additional proposed language constructs in combination with CogniCrypt$_{\text{HYB}}$. In chapter 5, we proposed additional language facilities for CrySL to tackle yet uncovered API usage constraint types (i.e., *Exception Handling*, *Controlling Method Call*, *Pre-Null-Check* and *Post-Null-Check*). However, we believe a simple integration in CrySL and consideration in CogniCrypt$_{\text{SAST}}$ will produce a high amount of reported warnings. In general, this is due to the fact CogniCrypt$_{\text{SAST}}$ inherently lacks runtime context and can therefore not decide in these cases whether a true or false positive is present. Although CogniCrypt$_{\text{SAST}}$ could still display all alleged API misuses in these cases, this would significantly reduce its usability. To overcome this, CogniCrypt$_{\text{SAST}}$ could be extended to CogniCrypt$_{\text{HYB}}$, a hybrid approach combining the advantages of static and dynamic analysis [LRCS14]. Thus, our proposed additional language facilities in CrySL could continue to be analyzed using static analysis to then validate the found API misuses by a dynamic analysis approach. Once the dynamic analysis confirms the API misuse, it gets reported as an actual API misuse to the user of CogniCrypt$_{\text{HYB}}$. Consequently, such an approach would heavily reduce false positives but indicating only relevant found API misuses. We believe that it is mandatory first to extend CogniCrypt$_{\text{SAST}}$ with a dynamic analysis approach and then implement our proposed language facilities in CrySL. For the precise implementation in CrySL, the data of the large-scale study could be used to validate our proposed improvements on the one hand and to be able to elaborate the formalities even more precisely on the other hand (e.g., possible corner cases we did not consider in our evaluation due to insufficient amount of data).

Usability of automatically generated API misuse reports by CogniCrypt$_{\text{SAST}}$. During our evaluation using CogniCrypt$_{\text{SAST}}$, we encountered two possible improvements for its usability.

The first improvement is related to the API misuse reports, which are generated automatically by CogniCrypt$_{\text{SAST}}$. During our evaluation, we experienced that the reports are sometimes hard to understand. Consider, for example, the code snippet shown in figure 6.2. It is expected to first instantiate the *FileReader* object through a call to the constructor *FileReader(file)*, to then *read()* arbitrary often the file's characters. At the end of the usage, there is a need to release the resource by calling *close()* (cf. CrySL rule in figure 5.1 without the consideration of our proposals for exception handling). Because *close()* is called twice in the code snippet of figure 6.2, an API misuse is correctly reported by CogniCrypt$_{\text{SAST}}$: *Unexpected call to method close on object of type java.io.FileReader*. In general, the user does not know about the full CrySL ruleset and its specifications. Therefore, we argue that the user may not know what is expected instead of calling *close()*. From the API misuse report, it is not clear that the

```
1 FileReader f = new FileReader("file"); // assume file exists
2 f.close();
3 f.close(); // API misuse
```

Figure 6.2: The code snippet shows an API misuse because *close()* is called twice.

second call to *close()* needs to be removed as no other method calls are allowed anymore. One approach would be to add more precise information to the automatically generated API misuse reports. Nevertheless, as the API designer/expert knows their API best, a better approach would be to extend CrySL in such a way that the API designer/expert can specify the API misuse report displayed in case of an API misuse. This could be extended by the results of the previously proposed large-scale study. The results can be used to adapt CogniCrypt$_{SAST}$ to suggest to the user the correct usage pattern automatically when an API misuse is indicated.

The second improvement is related to the overall report of API misuses by CogniCrypt$_{SAST}$. The users of static analysis tools tend not to use them if the rate of false positives is too high [BBC+10, LRCS14]. Currently, CogniCrypt$_{SAST}$ displays all found API misuses in a list, without considering how the list is sorted in terms of the likelihood of an API misuse to be a true positive. Since false positives cannot be ruled out with certainty, a ranking strategy by displaying first the identified API misuses that are most likely to be true positives [Ama18] could help to improve the usability of CogniCrypt$_{SAST}$. Again, the results from the large-scale study may be helpful because the database contains validated API misuses. CogniCrypt$_{SAST}$ could query the database to confirm the found API misuse. Once an API misuse is confirmed, the API misuse is most likely a true positive, causing its report to appear at the top of the list.

With the integration of both proposed improvements, we believe we can elevate CogniCrypt$_{SAST}$ to a new level of maturity.

Considering inheritance and extension of API specifications in CrySL. The last two improvements are related to the type-hierarchy of API classes and their library versions.

Inheritance and extension of API classes are common outside the context of cryptography but are not reflected in the language facilities of CrySL (cf. section 4.2.2). Currently, an API designer/expert must repeat himself when defining API usage constraints in the subclass' CrySL rule, which may have already been defined in the base class' rule. Therefore, the inheritance tree could also be reflected in the specification of CrySL rules to avoid unnecessarily repeating API usage constraints.

Furthermore, CrySL rules should be dependent on the framework version. Although this thesis' scope is limited to SE8[3], the API usage constraints may differ from version to version [LWY+18]. Thus, it is advisable to provide the API designer/expert possibilities to specify API usage constraints dependent on the API or library version. CogniCrypt$_{SAST}$ may then check the respective version to consume the appropriate CrySL ruleset.

In summary, we believe a large-scale study of API usage constraints, API misuses, and their different types would provide researchers a fundamental basis to improve and refine their API misuse detecting approaches and specification languages significantly. With a possible transfer of the data into *MUBench*, existing API misuse detecting approaches could be measured more meaningfully and ensure better comparability. In the case of CrySL and CogniCrypt$_{SAST}$, we proposed two topics for improvements. The first topic elevates the capabilities of CrySL and CogniCrypt$_{SAST}$ by implementing our proposed language facilities into CrySL in combination with a hybrid approach called CogniCrypt$_{HYB}$, to validate respective API misuses (e.g., in the case of *Exception Handling* violations). The second topic targets the overall usability of CogniCrypt$_{SAST}$ regarding its API misuse reports. With a possible integration of both, we believe CogniCrypt would elevate to a capable analysis tool also able to detect a variety of API misuses outside the context of cryptography. Besides, we found two more possible improvements in the case of CrySL. With the consideration of type-hierarchy and library versions, CrySL would become more flexible and convenient to use in specifying API usage constraints.

[3] https://docs.oracle.com/javase/8/docs/api/

7 Conclusion

In this thesis, we evaluated CrySL and CogniCrypt$_{SAST}$ on APIs outside the context of cryptography to assess their capabilities. This thesis is guided by five research questions (RQs). Each research question provides its own contribution to either CrySL and CogniCrypt$_{SAST}$ or to the general topic of API misuses and detection.

We first surveyed the literature. Almost every study presented different and inconsistent definitions for the conceptual levels of API directives, API usage constraints and API misuses. Therefore, we have introduced our own definitions for these conceptual levels but with consideration of previous work to answer **RQ1**. This allowed us to unify previous work related to API usage constraints (i.e., the root cause of API misuses) to form a comprehensive classification framework of such (**RQ2**). While classification frameworks already exist specifically for evaluating the capabilities of API misuse detectors (e.g., *MuC* [ANN+19]), these frameworks lack API usage constraint types. We have shown that our presented classification framework has three advantages compared to *MuC*. Firstly, it is more fine-granular with new API usage constraint types. Secondly, our classification framework allows the localization of API usage constraints and API misuses as each type is associated with the different parts of an API usage (i.e., an API method call). Finally, our classification framework respects the perspective of an API designer/expert providing a more natural view on API misuses.

With our classification framework, we have re-classified 88 API misuses in total, mainly from the *MUBench-Dataset* [ANN+16], and applied them to CrySL and CogniCrypt$_{SAST}$. The evaluation has shown that only 8 API usage constraints could successfully be specified and their respective API misuse detected. This is since CrySL and CogniCrypt$_{SAST}$ are tailored to the domain of cryptography, but APIs outside the context of cryptography impose different constraint types not covered by them. Thus, CrySL and CogniCrypt$_{SAST}$ are in general not applicable to non-cryptographic APIs without further improvements which answers **RQ3**. We were therefore also unable to evaluate precision and recall (**RQ4**).

We have presented three theoretical extensions for CrySL with which it would be possible to detect 93% of the API misuses we considered (**RQ5**). However, these extensions have the disadvantage that, most likely, many warnings will be displayed in practice, thus weakening the applicability of CogniCrypt$_{SAST}$. To counteract this abuse, we proposed approaches for ongoing work, such as a hybrid solution called CogniCrypt$_{HYB}$ taking advantage of dynamic and static analysis. However, it is advisable from the perspective of static specification languages like CrySL to further explore the different API usage constraint types in a large-scale study. This would help to better understand the different API usage constraint types and to use the data to make future evaluations of API misuse detectors more meaningful.

8 Appendix

Table 8.1: Mapping of API directive types and API misuse categories elaborated by Monperrus et al. [METM12], Amann [Ama18] and Li [Li20], to the API usage constraint types of our classification framework. Their categories annotated with an asterisk do not perfectly match our definition of an API usage constraint or overlap with another API usage constraint type. Note that their categories may map to more than one of our introduced types.

API Usage Constraint Types	**Mapping of**		
	API Directive Types	**API Misuse Categories**	
	Monperrus et al.	Amann	Li
Receiver			
Post-Call(s)	Post-Call Directive	n.a.	n.a.
Post-Null-Check	n.a.	Missing and Redundant Null Checks*	Missing and Redundant Null Checks*
Method Call			
Method Call Sequence	Method Call Sequence Directive, Non Call-based State Directive*, Miscellaneous Method Call Directives*	Missing Method Calls, Redundant Method Calls*, Missing Iteration*, Redundant Iteration*	Missing and Redundant API Call*, Replaced Name*
Conditional Method Call	n.a.	Missing value and State Condition, Missing Iteration, Redundant Iteration*	Missing Object State*, Missing Return Value*
Forbidden Method Call	Method Call Visibility Directive*	Redundant Method Calls*	n.a.
Passed Arguments			Missing Return Value*, Missing Object State*, Rep Args*

Argument State	n.a.	Missing Value and State Condition*, Redundant Value and State Condition*	Missing Return Value*, Missing Object State*
String Format	String Format Directive		n.a.
Number Range	Number Range Directive		
Argument Correlation	Method Parameter Correlation Directive		
Argument Type	Method Parameter Type Directive	n.a.	
Pre-Null-Check	Not Null Directive	Missing and Redundant Null Checks*	Missing and Redundant Null Checks*
Multiple Assignments			
Exception Handling	Exception Raising Directive	Missing and Redundant Exception Handling	Exception
Context	Method Call Visibility Directive*	Missing Context Conditions*, Redundant Context Conditions*	n.a.
Synchronization	Synchronization Directive	Missing Synchronization Conditions, Redundant Synchronization Conditions*	Missing Synchronization, Redundant Synchronization*
Threading	Method Call Visibility Directive*	Missing Context Conditions*, Redundant Context Conditions*	n.a.
High-Level Constraints	n.a.	n.a	n.a

Protocol API Misuse Evaluation

Overall Information

Date:	24.11.2020		
API:	java.swing.JFrame	Time for Evaluation:	1:00h
Example from:	MUBench	Evaluation finished:	true
Classification by MuC/Li:	missing method call	Classified by our framework:	Method Call Sequence
Overlapping with:	-		
Result CrySL:	**specifiable**		
Result CogniCrypt$_{SAST}$:	**not working**		

Info:
- This is the prime example of a sub-sequence! Maybe working when bug gets fixed.

Misuse Pattern

```
1 JFrame f = new JFrame("Main Window");
2 f.setVisible(true);
```

Correct Pattern

```
1 JFrame f = new JFrame("Main Window");
2 f.pack();
3 f.setVisible(true);
```

From MUBench: Calls JFrame.setVisible(true) without calling *JFrame.pack()*, such that the frame is not layouted before being shown. Also calling *pack()* after *setVisible(true)* may lead to undesired effects, such as the window being moved to the default platform location.

Reference to API specification: Link

Problems regarding CrySL

This particular scenario is different from the others. *setVisible(true)* requires to call *pack()* beforehand and not the other way around. The only way to specify this scenario in CrySL is to model all valid sequences starting from the invocation of pack() until calling *setVisisible(true)*. This may bloat the regular expression (`ORDER` section). The boolean is no problem here as we can specify this constraint in the `CONSTRAINT` section.

Problems regarding CogniCrypt$_{SAST}$

Although we did not specify all possible sequences, detecting a misuse is theoretically possible regarding a violation of the correct pattern above (when bug gets fixed).

Additional Information

-

Suggestions for Improvements

It would be questionable whether a blacklisting approach would be better in such a scenario. It is a general question how often such processes occur in practice. We need more data!

Figure 8.1: Example of creating a protocol for an API misuse in the evaluation process.

Bibliography

[ABKT16] K. Allix, T. F. Bissyandé, J. Klein, and Y. L. Traon. Androzoo: Collecting millions of android apps for the research community. In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, pages 468–471, Austin, TX, USA, 2016. https://doi.org/10.1145/2901739.2903508.

[Ama18] Sven Amann. *A Systematic Approach to Benchmark and Improve Automated Static Detection of Java-API Misuses.* Ph.D. thesis, Department of Computer Scienece at the technical university of Darmstadt, March 2018. https://tuprints.ulb.tu-darmstadt.de/7422/.

[ANN+16] Sven Amann, Sarah Nadi, Hoan A. Nguyen, Tien N. Nguyen, and Mira Mezini. Mubench: A benchmark for api-misuse detectors. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, page 464–467, New York, NY, USA, 2016. Association for Computing Machinery. https://doi.org/10.1145/2901739.2903506.

[ANN+19] 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. https://doi.org/10.1109/TSE.2018.2827384.

[ASN17] Anna G. Troshina Sergey N. Vasiliev Alexander S. Novikov, Alexey N. Ivutin. The approach to finding errors in program code based on static analysis methodology. *2017 6th Mediterranean Conference on Embedded Computing*, pages 1–4, 2017. https://doi.org/10.1109/MECO.2017.7977127.

[BBA09] Kevin Bierhoff, Nels E. Beckman, and Jonathan Aldrich. Practical api protocol checking with access permissions. In Sophia Drossopoulou, editor, *ECOOP 2009 – Object-Oriented Programming*, pages 195–219, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-03013-0_10.

[BBC+10] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: Using static analysis to find bugs in the real world. *Commun. ACM*, 53(2):66–75, February 2010. https://doi.org/10.1145/1646353.1646374.

[BBG+63] J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, P. Naur, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, and M. Woodger. Revised report on the algorithmic language ALGOL 60. *The Computer Journal*, 5(4):349–367, 01 1963. https://doi.org/10.1093/comjnl/5.4.349.

[BKA11] Nels E. Beckman, Duri Kim, and Jonathan Aldrich. An empirical study of object protocols in the wild. In Mira Mezini, editor, *ECOOP 2011 – Object-Oriented Programming*, pages 2–26, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-22655-7_2.

[BKEI09] Oren Ben-Kiki, Clark Evans, and Brian Ingerson. Yaml ain't markup language (yaml™) version 1.1. *Working Draft 2008-05*, 11, 2009. https://yaml.org/.

[BMM09] Marcel Bruch, Martin Monperrus, and Mira Mezini. Learning from examples to improve code completion systems. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, page 213–222, New York, NY, USA, 2009. Association for Computing Machinery. https://doi.org/10.1145/1595696.1595728.

[BMM10] M. Bruch, M. Mezini, and M. Monperrus. Mining subclassing directives to improve framework reuse. In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, pages 141–150, 2010. https://doi.org/10.1109/MSR.2010.5463347.

[CNKX16] Alexia Chatzikonstantinou, Christoforos Ntantogian, Georgios Karopoulos, and Christos Xenakis. Evaluation of cryptography usage in android applications. In *Proceedings of the 9th EAI International Conference on Bio-Inspired Information and Communications Technologies (Formerly BIONETICS)*, BICT'15, page 83–90, Brussels, BEL, 2016. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering). https://doi.org/10.4108/eai.3-12-2015.2262471.

[CPNM18] Ervina Cergani, Sebastian Proksch, Sarah Nadi, and Mira Mezini. Investigating order information in api-usage patterns: A benchmark and empirical study. In *ICSOFT*, pages 91–102, 2018. https://doi.org/10.5220/0006839000570068.

[DH09] U. Dekel and J. D. Herbsleb. Improving api documentation usability with knowledge pushing. In *2009 IEEE 31st International Conference on Software Engineering*, pages 320–330, 5000 Forbes Avenue, Pittsburgh, PA 15213 USA, 2009. https://doi.org/10.1109/ICSE.2009.5070532.

[DWPJV02] Bart De Win, Frank Piessens, Wouter Joosen, and Tine Verhanneman. On the importance of the separation-of-concerns principle in secure software engineering. In *Workshop on the Application of Engineering Principles to System Security Design*, pages 1–10, Celestijnenlaan 200A, B-3001 Heverlee, Belgium, 2002.

[EBFK13] 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 Communications Security*, CCS '13, pages 73–84, New York, NY, USA, 2013. Association for Computing Machinery. https://doi.org/10.1145/2508859.2516693.

[GWL+19] Z. Gu, J. Wu, J. Liu, M. Zhou, and M. Gu. An Empirical Study on API-Misuse Bugs in Open-Source C Programs. In *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, volume 1, pages 11–20. IEEE, 2019. https://doi.org/10.1109/COMPSAC.2019.00012.

[HJZ13] K. Herzig, S. Just, and A. Zeller. It's not a bug, it's a feature: How misclassification impacts bug prediction. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 392–401, 2013. https://doi.org/10.1109/ICSE.2013.6606585.

[HP04] David Hovemeyer and William Pugh. Finding Bugs is Easy. *SIGPLAN Not.*, 39(12):92–106, December 2004. https://doi.org/10.1145/1052883.1052895.

[Isl20] Md Johirul Islam. *Towards understanding the challenges faced by machine learning software developers and enabling automated solutions.* Ph.D. thesis, Iowa State University, 2020. https://doi.org/10.31274/etd-20200902-68.

[JCX19] Z. Jin, K. Y. Chee, and X. Xia. What Do Developers Discuss about Biometric APIs? In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 348–352, 2019. https://doi.org/10.1109/ICSME.2019.00053.

[JJE14] René Just, Darioush Jalali, and Michael D. Ernst. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 437–440, New York, NY, USA, 2014. Association for Computing Machinery. https://doi.org/10.1145/2610384.2628055.

[JSMB13] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge. Why don't software developers use static analysis tools to find bugs? In *2013 35th International Conference on Software Engineering (ICSE)*, pages 672–681, 2013. https://doi.org/10.1109/ICSE.2013.6606613.

[KDP+19] 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. https://doi.org/10.1145/3293882.3330552.

[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, pages 931–936. IEEE Press, 2017. https://doi.org/10.1109/ASE.2017.8115707.

[Kra99] Douglas Kramer. Api documentation from source code comments: A case study of javadoc. In *Proceedings of the 17th Annual International Conference on Computer Documentation*, SIGDOC '99, page 147–153, New York, NY, USA, 1999. Association for Computing Machinery. https://doi.org/10.1145/318372.318577.

[KS14] Maria Kechagia and Diomidis Spinellis. Undocumented and unchecked: Exceptions that spell trouble. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, page 312–315, New York, NY, USA, 2014. Association for Computing Machinery. https://doi.org/10.1145/2597073.2597089.

[KSA+18] S. Krüger, J. Späth, K. Ali, E. Bodden, and M. Mezini. CrySL: An Extensible Approach to Validating the Correct Usage of Cryptographic APIs. *ECOOP 2018*, (No. 10):pp. 10:1–10:27, 2018. https://doi.org/10.4230/LIPIcs.ECOOP.2018.10.

[KTBR20] Raffi Khatchadourian, Yiming Tang, Mehdi Bagherzadeh, and Baishakhi Ray. An empirical study on the use and misuse of java 8 streams. In *23rd International Conference, FASE 2020*, pages 97–118. ETAPS 2020, April 2020. https://doi.org/10.1007/978-3-030-45234-6.

[LBLH11] Patrick Lam, Eric Bodden, Ondrej Lhotak, and Laurie Hendren. The soot framework for java program analysis: a retrospective. In *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, Oktober 2011. https://tubiblio.ulb.tu-darmstadt.de/59322/.

[LCWZ14] David Lazar, Haogang Chen, Xi Wang, and Nickolai Zeldovich. Why does cryptographic software fail? a case study and open problems. In *Proceedings of 5th Asia-Pacific Workshop on Systems*, APSys '14, New York, NY, USA, 2014. Association for Computing Machinery. https://doi.org/10.1145/2637166.2637237.

[Li20] Xia Li. *An Integrated Approach for Automated Software Debugging via Machine Learning and Big Code Mining.* Ph.D. thesis, The University of Texas at Dellas, 2020. https://hdl.handle.net/10735.1/8945.

[LLS+18] H. Li, S. Li, J. Sun, Z. Xing, X. Peng, M. Liu, and X. Zhao. Improving api caveats accessibility by mining api caveats knowledge graph. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 183–193, 2018. https://doi.org/10.1109/ICSME.2018.00028.

[LLY+20] Tao Lv, Ruishi Li, Yi Yang, Kai Chen, Xiaojing Liao, XiaoFeng Wang, Peiwei Hu, and Luyi Xing. Rtfm! automatic assumption discovery and verification derivation from library document for api misuse detection. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, page 1837–1852, New York, NY, USA, 2020. Association for Computing Machinery. https://doi.org/10.1145/3372297.3423360.

[LRCS14] Kaituo Li, Christoph Reichenbach, Christoph Csallner, and Yannis Smaragdakis. Residual investigation: Predictive and precise bug detection. *ACM Trans. Softw. Eng. Methodol.*, 24(2), December 2014. https://doi.org/10.1145/2656201.

[LWY+18] Tianyue Luo, Jingzheng Wu, Mutian Yang, Sizhe Zhao, Yanjun Wu, and Yongji Wang. Mad-api: Detection, correction and explanation of api misuses in distributed android applications. In Marco Aiello, Yujiu Yang, Yuexian Zou, and Liang-Jie Zhang, editors, *Artificial Intelligence and Mobile Services – AIMS 2018*, pages 123–140, Cham, 2018. Springer International Publishing. https://doi.org/10.1007/978-3-319-94361-9.

[LZL+14] Qingzhou Luo, Yi Zhang, Choonghwan Lee, Dongyun Jin, Patrick O'Neil Meredith, Traian Florin ŞerbănuŢă, and Grigore Roşu. 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. https://doi.org/10.1007/978-3-319-11164-3_24.

[METM12] Martin Monperrus, Michael Eichberg, Elif Tekes, and Mira Mezini. What should developers be aware of? an empirical study on the directives of api documentation. *Empirical Software Engineering*, 17(6):703–737, Dec 2012. https://doi.org/10.1007/s10664-011-9186-4.

[MR13] W. Maalej and M. P. Robillard. Patterns of knowledge in api reference documentation. *IEEE Transactions on Software Engineering*, 39(9):1264–1282, September 2013. https://doi.org/10.1109/TSE.2013.12.

[NHKO20] Sebastian Nielebock, Robert Heumüller, Jacob Krüger, and Frank Ortmeier. Cooperative api misuse detection using correction rules. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results*, ICSE-NIER '20, page 73–76, New York, NY, USA, 2020. Association for Computing Machinery. https://doi.org/10.1145/3377816.3381735.

[NHO18] Sebastian Nielebock, Robert Heumüller, and Frank Ortmeier. Commits as a basis for api misuse detection. In *Proceedings of the 7th International Workshop on Software Mining*, SoftwareMining 2018, page 20–23, New York, NY, USA, 2018. Association for Computing Machinery. https://doi.org/10.1145/3242887.3242890.

[NKMB16] 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, New York, NY, USA, 2016. Association for Computing Machinery. https://doi.org/10.1145/2884781.2884790.

[NNP+09] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, Jafar M. Al-Kofahi, and Tien N. Nguyen. Graph-based mining of multiple object usage patterns. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, pages 383–392, New York, NY, USA, 2009. Association for Computing Machinery. https://doi.org/10.1145/1595696.1595767.

[NVN19] Tam The Nguyen, Phong Minh Vu, and Tung Thanh Nguyen. Api misuse correction: A statistical approach. *arXiv preprint arXiv:1908.06492*, 2019. https://arxiv.org/abs/1908.06492.

[Oraa] Oracle. NullPointerException documentation. online. https://docs.oracle.com/javase/8/docs/api/java/lang/NullPointerException.html, accessed on 27.02.21.

[Orab] Oracle. Swing's threading policy. online. https://docs.oracle.com/javase/8/docs/api/javax/swing/package-summary.html, accessed on 04.12.20.

[PG12] M. Pradel and T. R. Gross. Leveraging test generation and specification mining for automated bug detection without false positives. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 288–298. IEEE, June 2012. https://doi.org/10.1109/ICSE.2012.6227185.

[Rat13] Martin P. Robillard; Eric Bodden; David Kawrykow; Mira Mezini; Tristan Ratchford. Automated api property inference techniques. *IEEE Transactions on Software Engineering*, 39(5):613–637, May 2013. https://doi.org/10.1109/TSE.2012.63.

[RSX+20] Xiaoxue Ren, Jiamou Sun, Zhenchang Xing, Xin Xia, and Jianling Sun. Demystify official api usage directives with crowdsourced api misuse scenarios, erroneous code examples and patches. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 925–936, New York, NY, USA, June 2020. Association for Computing Machinery. https://doi.org/10.1145/3377811.3380430.

[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 '19, pages 2455–2472, New York, NY, USA, 2019. Association for Computing Machinery. https://doi.org/10.1145/3319535.3345659.

[RYX+20] X. Ren, X. Ye, Z. Xing, X. Xia, X. Xu, L. Zhu, and J. Sun. Api-misuse detection driven by fine-grained api-constraint knowledge graph. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 461–472, 2020. https://doi.org/10.1145/3324884.3416551.

[SB15] Yannis Smaragdakis and George Balatsouras. Pointer analysis. *Foundations and Trends® in Programming Languages*, 2:1–69, April 2015. https://doi.org/10.1561/2500000014.

[SBLV+19] Simone Scalabrino, Gabriele Bavota, Mario Linares-Vásquez, Michele Lanza, and Rocco Oliveto. Data-driven solutions to detect api compatibility issues in android: an empirical study. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 288–298. IEEE, May 2019. https://doi.org/10.1109/MSR.2019.00055.

[SHA15] J. Sushine, J. D. Herbsleb, and J. Aldrich. Searching the state space: A qualitative study of api protocol usability. In *2015 IEEE 23rd International Conference on Program Comprehension*, pages 82–93, May 2015. https://doi.org/10.1109/ICPC.2015.17.

[SSD15] M. A. Saied, H. Sahraoui, and B. Dufour. An observational study on api usage constraints and their documentation. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 33–42, March 2015. https://doi.org/10.1109/SANER.2015.7081813.

[Ste20] Stefan Krüger. *CogniCrypt - The Secure Integration of Cryptographic Software*. Ph.D. thesis, University Paderborn, October 2020. https://doi.org/10.17619/UNIPB/1-1039.

[Tai96] Antero Taivalsaari. On the notion of inheritance. *ACM Comput. Surv.*, 28(3):438–479, September 1996. https://doi.org/10.1145/243439.243441.

[VRGH+00] Raja Vallée-Rai, Etienne Gagnon, Laurie Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. Optimizing java bytecode using the soot framework: Is it feasible? In David A. Watt, editor, *Compiler Construction, 9th International Conference*, pages 18–34, Berlin, Heidelberg, March 2000. ETAPS 2000, Springer Berlin Heidelberg. ISBN 978-3-540-46423-5.

[WLW+19] M. Wen, Y. Liu, R. Wu, X. Xie, S. Cheung, and Z. Su. Exposing library api misuses via mutation analysis. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 866–877, May 2019. https://doi.org/10.1109/ICSE.2019.00093.

[WRE+19] A. K. Wickert, M. Reif, M. Eichberg, A. Dodhy, and M. Mezini. A dataset of parametric cryptographic misuses. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 96–100, May 2019. https://doi.org/10.1109/MSR.2019.00023.

[WWL+18] Ying Wang, Ming Wen, Zhenwei Liu, Rongxin Wu, Rui Wang, Bo Yang, Hai Yu, Zhiliang Zhu, and Shing-Chi Cheung. Do the dependency conflicts in my project matter? In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, pages 319–330, New York, NY, USA, 2018. Association for Computing Machinery. https://doi.org/10.1145/3236024.3236056.

[WZL07] 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. https://doi.org/10.1145/1287624.1287632.

[ZM19] H. Zhong and H. Mei. An empirical study on api usages. *IEEE Transactions on Software Engineering*, 45(4):319–334, December 2019. https://doi.org/10.1109/TSE.2017.2782280.