FUM - A Framework for API Usage constraint and Misuse Classification

1st Michael Schlichtig [3]
Heinz Nixdorf Institute, Paderborn University
Paderborn, Germany
michael.schlichtig@uni-paderborn.de

2nd Steffen Sassalla [3]
Hasso Plattner Institute, University of Potsdam, Germany
steffen.sassalla@student.hpi.de

3rd Krishna Narasimhar [3]
Technische Universität Darmstadt
Darmstadt, Germany
kri.nara@cs.tu-darmstadt.de

4th Eric Bodden [3]
Heinz Nixdorf Institute, Paderborn University & Fraunhofer IEM
Paderborn, Germany
eric.bodden@uni-paderborn.de

Abstract—Application Programming Interfaces (APIs) are the primary mechanism that developers use to obtain access to third-party algorithms and services. Unfortunately, APIs can be misused, which can have catastrophic consequences, especially if the APIs provide security-critical functionalities like cryptography. Understanding what API misuses are, and for what reasons they are caused, is important to prevent them, e.g., with API misuse detectors. However, definitions and nominations for API misuses and related terms in literature vary and are diverse. This paper addresses the problem of scattered knowledge and definitions of API misuses by presenting a systematic literature review on the subject and introducing FUM, a novel Framework for API Usage constraint and Misuse classification. The literature review revealed that API misuses are violations of API usage constraints. To capture this, we provide unified definitions and use them to derive FUM. To assess the extent to which FUM aids in determining and guiding the improvement of an API misuses detectors’ capabilities, we performed a case study on CogniCrypt, a state-of-the-art misuse detector for cryptographic APIs. The study showed that FUM can be used to properly assess CogniCrypt’s capabilities, identify weaknesses and assist in deriving mitigations and improvements. And it appears that more generally FUM can aid the development and improvement of misuse detection tools.

Index Terms—API misuses, API usage constraints, classification framework, API misuse detection, static analysis

I. INTRODUCTION

Software reuse is one of the fundamental principles of good software development [24]. To facilitate such reuse, developers can take advantage of previously written functionality exposed as application programming interfaces (APIs). For example, consider Java that comes bundled with Java Class Library (JCL) [61] - 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. Besides language maintainers such as Oracle that provide standard APIs, there is also the community of developers that supports the idea of sharing functionality, e.g., by providing APIs as free or open-source software, e.g., the Maven repository [46] consists of thousands of APIs provided by a huge community that a developer can benefit from using. On average, a Java project depends on 14 different libraries [79]. Consequently, developers often are composers 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 [22], [28], [40], [51], [68], such as the lack of appropriate documentation, the complexity of the API, or even an inadequate level of abstraction [51]. As a result, developers tend to incorrectly integrate APIs in their code which leads to misbehavior, errors, or even program crashes. Research [2], [3], [22], [43] has shown that the misuse of APIs goes beyond specific and complex domains like cryptographic APIs, misuses occur all over API usages. Amann et al. [3] examined the prevalence of API misuses in several bug datasets [29], [31]. They found that 89 of all 1189 reviewed bugs were API misuses (7.49%), however, 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 [42] performed a systematic and extensive empirical study of API misuses mining GitHub [20]. They found that 50.6% of all bug-fixing commits between 2011 and 2018 were related to API misuses. Misuses of APIs like the JCA are quite common [51] and tend to have catastrophic effects because of their security-critical nature, even if they do not cause crashes. Moreover, even if high-quality documentation is provided, this may still not be enough for developers to avoid API misuses altogether [1]. To reduce API misuses developers need more support, e.g., in the form of API misuse detectors [1], [17], [26], [37], [49], [50], [52], [53], [68], [69], [75].

Using misuse detectors can be very beneficial for developers. However, they are often infrequently adopted in practice, because of lack of analysis quality (e.g., precision and recall), usability issues [12], and the Achilles heel of static analysis: false positives [30]. Among other things, a main contributing...
factor to this is the quality of reporting which, in turn, originates from a lack of standardized taxonomy of API misuse classifications, their severity, etc. The acuteness of this problem was discussed by Robillard et al. [71] who surveyed a decade of studies on API misuses where the authors state “we observe a lack of uniformity in terminology and definitions for the properties inferred by various approaches“. Although they observe the need for uniformity, their survey was aimed at understanding and categorizing the API usage inference techniques and not to standardize the misuse classification terminologies themselves. The first approach for this was provided by Amann et al. [1] who studied API misuses in the wild, classified them, and introduced a taxonomy called API-Misuse Classification framework (MuC). However, they derived MuC empirically from API misuses instead of building the taxonomy solely based on a theoretical foundation.

This paper presents a literature survey on API misuses and classification taxonomies, to understand the state of the art and to derive required definitions and an API misuse classification framework. Specifically, we exhaustively studied the previous work of Amann et al. [1] with MuC, refinements that Li [42] made to MuC, and the work by Monperrus et al. [47] who categorized the different types of API directives in API documentation. To derive a concrete classification framework, this work focuses on API misuses in Java programs. Java is one of the most popular languages [78] and many of its features are representative of those in other commonly used languages. Moreover, many previous studies on API misuses have been performed on Java APIs, which gives us the ideal starting point for both comparison and expansion.

To arrive at our taxonomy, we first conducted a scientific literature study, seeking to answer the following research question:

- a) RQ1: How is the term API misuse defined, delimited, and made tangible?

To improve the development of analysis tools for API misuse detection, it is necessary to understand what API misuses are and how they are caused. This requires a theoretical foundation with clear definitions and an exploration of the sources of API misuses inside the program, i.e., to identify the nature of API misuses from the perspective of the API designer/expert. We conducted a systematic literature study surveying to what extent related work already provides an appropriate framework. We found that, while relevant related work exists, no existing work on the subject provides an appropriate framework.

- b) RQ2: What does an API classification framework need to look like that links API misuses to their causes and the respective part of the method call?

The paper overall is structured as follows. We first provide background on APIs and show an example of their (mis)use. In section III we provide an overview of previous works that have declared diverse definitions related to API misuses, elaborated on the root causes, and focused on classifying the different misuse types. As a result, in section IV we introduce our own definitions with respect to previous research, originating from the results of RQ1. Based on this theoretical foundation, we then address RQ2 by contributing FUM a comprehensive „Framework for API Usage constraint and Misuse classification“ (section V). We further clarify this in section VI by discussing the differences between FUM and other classification frameworks. Section VII then presents a case study that applies FUM to the crypto-API misuse detector CogniCrypt which is considered state of the art in API misuse detection and shown to be able to cover a majority of cryptographic API misuses and has better precision compared to other cryptographic API misuse detectors [8], [18], [23]. The study shows that FUM assists one in classifying API usage constraints, API misuses, and in improving API misuse detectors. Section VIII concludes.

II. Background

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 [47]. The private implementation is the part that implements the functionality of the library. However, concrete implementation details are usually not exposed to the developer (i.e., to the library’s user). Further on we will not 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 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 [80]. 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).

Consider, for instance, a correct application of the java.io.FileReader [57]. Whenever a file is successfully opened, the file’s content can be read() and following its usage, the resource needs to be released by invoking close(). Consequently, it is a misuse not to call close() at the end of every usage of FileReader.
To support developers, APIs are delivered with documentation, which can be very diverse in nature \cite{47}. For example, parts of the documentation can be related to typical use case scenarios, code snippets, constraints, or even performance.

We now have a better 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 which we address in the following sections.

III. RELATED WORK - A SURVEY OF DEFINITIONS AND PERSPECTIVES

This section addresses \textbf{RQ1} providing an overview of several studies focused on different aspects of API documentation, misuses, and even beyond. We performed a systematic literature review \cite{35}. As part of our literature survey, we used keywords "API misuses", "misuse detection", "library misuse" among other related terms in Google scholar and IEEE explore in the time from November 2020 until February 2021. To complete the results of the literature research process, we have snowballed relevant results backward and forwards. Overall, we considered 69 publications in discussion with two or more co-authors to filter out outcomes. Furthermore, we also actively engaged in discussions with the authors of previous studies. The most relevant publications are presented in this paper.

Whenever an API is used, developers need comprehensive and explanatory documentation. Past research focused on the different kinds of API documentation. We consider previous work by its range on different (implicit) types of API documentation and its noncompliance. For example, Robillard and Maalej \cite{45} focused on every part of an API documentation, whereas Lv et al. \cite{44} only considered those parts of documentation for which noncompliance would result in errors or misbehavior. Further, we also show that almost every study provides its own definitions (denoted in bold letters).

A. API Documentation Types and Definitions

Dekel and Herbsleb \cite{13} researched how developers’ awareness of usage directives could be increased. They defined usage directives as parts of API documentation that capture nontrivial, infrequent, and possibly unexpected information, introducing ten different types of usage directives, e.g., Restrictions, which address the context in which the API method should (not) be called. They developed eMoose, an Eclipse \cite{15} plugin to highlight usage directives in the IDE.

Bruch et al. \cite{9} investigated parts of API documentation focussing on extensibility. They defined parts of API documentation related to extensibility and inheritance as subclassing directives, e.g., the subclassing directive subclasses may extend this method requires subclasses to call the super method. In total, they introduced four types of subclassing directives.

Based on both studies, Monperrus et al. \cite{47} 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 \cite{61}, JFace \cite{16}, and the Apache Commons Collections \cite{4}. Analyzing 4,561 API elements (i.e., documentation of interfaces, packages, classes, etc.) in total, they showed the prevalence of each type, respectively.

Robillard and Maalej \cite{45} empirically elaborated a comprehensive taxonomy of API knowledge types which are patterns of knowledge classifying a specific part of API documentation, e.g., the type Directive specifies what developers are (not) allowed to do with the API. Further, the Quality type describes non-functional requirements like performance implications. In total, they introduced twelve of these types.

B. API Usage Constraint Types and Definitions

In this section, we present studies on API documentation subsets that a developer must comply with to avoid encountering misbehavior, errors, or vulnerabilities.

Li et al. \cite{41} narrowed down the set of API documentation parts to API caveats which are directives where noncompliance would likely incur unexpected program behaviors or errors. Moreover, they introduced a classification framework of syntactic patterns that comprises ten types of API caveats.

Lv et al. \cite{44} considered only the parts of API documentation for which noncompliance would result in severe consequences, so-called integration assumptions (IAs). Each IA has its constraint related to pre-conditions (e.g., parameter length limit), post-conditions, or the invocation context.

Similarly, Nguyen et al. \cite{54} classified constraint types that lead to serious programming errors but without providing a concrete definition. They categorized such constraints as temporal order (e.g., API method calls are expected in a particular order), pre-conditions and post-conditions, argument value, and finally, exception (e.g., handling of certain exceptions). They proposed a novel approach called Statistical Approach for API Misuses (SAM) that tries to detect noncompliance.

Saied et al. \cite{7} investigated so-called API usage constraints and found that 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 results in failures), Nullness allowed (i.e., passing null value has a certain semantics), Range limitation (i.e., restricted numeric value), and Type restriction (i.e., restricted parameter type). Except Nullness allowed all introduced types restrict 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.

Addressing this problem, Amann \cite{2} 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. Based on the comprehensive classification framework of API directives of Monperrus et al.’s \cite{47} they provided an overview of API usage constraints categories.
In contrast, Ren et al.’s definition also comprises parts of the API documentation in which noncompliance does not necessarily result in errors or misbehavior. They defined API usage directives as “contracts, constraints, and guidelines that specify what developers are allowed/not allowed to do with the API”.

In addition, several (mostly empirical) studies investigate only very certain constraints, mostly referred to as object protocols which according to Beckman et al. are “[... ] dictating the ordering of method calls on objects of a particular class”. An example of a predefined method call order is the java.util.FileReader where read() is only allowed to be called if the resource was opened successfully.

C. 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 1, API usage constraints are a subset of API directives for which violation results in misbehavior, errors, or vulnerabilities at runtime (cf. section III-B). The dashed line splits API directives and API usage constraints as 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, 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 IV.

The classification, detection, and mitigation of API misuses have been studied in various specific fields of research – for instance, cryptography [17], 39, 40, 51, machine learning [28], stream APIs [34], and even biometric APIs [29], as well as in the overall analysis of API misuses [2], 27, 32, 33, 42, 73. Further, API misuses are not specific to just one programming language like Java. There are also studies focusing on API misuses in different languages, e.g., C [22] or Python [28]. Thus, API misuse is a more general problem for which we observed different kinds of definitions in literature.

Khatchadourian et al. [34] studied how Stream APIs are misused. They defined 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 commits. They derived a classification framework comprising 15 different categories.

Especially in the domain of cryptography, researchers are interested in API misuses [11], 17, 59, 40, 51. Nadi et al. [51] found that cryptographic APIs’ low level of abstraction leads to incorrect usage. As a result, APIs become 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. [17] conducted an empirical study on Android apps. They found that 88% of all apps have at least one mistake using cryptographic APIs. They showed that an API misuse does not necessarily result in errors or crashes but sensitive vulnerabilities causing possible security breaches. However, they did not provide any precise definition of an API misuse.

According to a study of Amann [2] in 2018, they were the first to systematically define the problem space of API misuses and empirically investigate the prevalence of API misuses in bug datasets. They introduced a comprehensive classification framework of 16 different API misuse categories: MuC [1]. They defined an API misuse as a violation of an (implicit) API usage constraint (cf. section III-B). Besides summarizing the findings of Monperrus et al. [47], Amann based their framework on the results from investigating about 1,200 bug reports from bug datasets like BugClassify [29] or Defect4J [31] and Monperrus et al.’s work. They identified a total of 164 API misuses. In addition, they provided their own dataset with pre-classified API misuses based on MuC, the so-called MUBench Dataset [48]. Together with MUBench is a benchmark to assess API misuse detectors - they provide a versatile toolbox in the field of API misuse and detection.

Since MUBench is based on API misuses from bug datasets, it might be limited. Thus, Li [42] conducted the first large-scale empirical study of API misuses in 2020. They defined “[...] API misuses 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 between 2011 and 2018. 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.

IV. Definitions

As discussed, several studies provide fundamental theory. They already present comprehensive classification frameworks of the different types of usage constraints and misuses [2], [6], 42, 47. 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.

Reason 1 — Dispersed theory knowledge across studies

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 [6], [7], [10], [76], while others try to classify - sometimes roughly - the spectrum of API directives, usage constraints, or API misuses [2], [7], [13], [27], [32]–[34], [41]–[45], [47], [54], [71]–[73], [76], [80] (cf. section III). The most comprehensive summary of relevant misuse types is yet provided by MuC [2]. However, the high abstraction level they chose misses several important pieces of information. For instance, Monperrus et al. [47] mentioned the Method Parameter Type Directive which mandates the allowed type of method parameters. For example, the compareTo(Date) method of java.sql.Timestamp [57] states: “Compares this Timestamp object to the given Date, which must be a Timestamp object.” Although mentioned by Amann, this type is not part of their API misuses’ derivation. Our approach ensures such root causes of API misuses are not missed. Essentially, we define all necessary terms surrounding API misuses (RQ1) based on our survey. This allows us to collect previous contributions, to provide one comprehensive framework (RQ2).

Reason 2 — Localization of usage constraints in method calls

We show that usage constraints can be associated with specific parts of a method call. For instance, usage constraints of type Post-Call Directive [47] categorize 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 passed arguments, and the context in which the API is used. Such localization helps to better understand the causes and types of API misuses. There is no previous taxonomy of API usage constraint types based on their location in API method calls.

A. Definition of “API”

We provided a definition for APIs in natural language based on Monperrus et al.’s work [47] (cf. section II) where we can basically divide APIs into two areas in terms of their application domain: (i) APIs with a focus on a particular domain. For example, cryptographic APIs implement cryptographic algorithms and provide basic functionality for data security. (ii) Other APIs are domain-independent. Consider, for instance, java.lang.String [65]. It provides overall functionality to work with strings that are used domain-independently. Note, this differentiation is important because oftentimes domain-specific APIs have more usage constraints and disclose very specific usage constraint types which we will see for the case study (cf. section VII) for the domain of cryptography. This perspective on APIs leads to the following definition:

Definition IV.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.

B. Definition of “API Directive”

As previous studies have shown [51], developers struggle to understand the intended usage of an API. Therefore, the API documentation is a core requirement that provides code examples, conceptual overviews, definitions of terms, programming guidelines, known bugs, and documented workarounds [36]. In fact, API documentation is manifold in its nature [45], [47]. Essential for developers are contracts enforced by the API. Such contracts, also called API directives, describe what developers are (not) allowed to do [45]. Monperrus et al. [47] 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, documentation does not necessarily need to be recorded in high quality, and thus it may be incomplete or contradictory [54], [71], [72]. It is not uncommon for documentation to be completely unavailable. Furthermore, some APIs take domain knowledge for granted [51] and miss documenting such domain-specific constraints. 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 [47] work leads to the following definition:

Definition IV.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 expected domain-specific knowledge.

C. Definition of “API Usage Constraint”

API directives can also be guidelines (e.g., hints for performance improvements). Hence, we can divide API directives further into usage constraints. 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 [60], the method next() may only be called if the iterator contains at least one element. At compile time, an API designer cannot force a developer to adhere to these constraints because the programming language does not provide any language constructs to ensure compliance (e.g., correct typing enforced by the compiler [2]). Overall, this implies the following definition of an API usage constraint based on the definition of Amann [2]:

Definition IV.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 per-
spective of an API designer/expert. They are 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.

D. Definition of “API Misuse”

API usage constraints can be violated, e.g., because
of a developer’s lack of domain knowledge [51] or improper
documentation [71]. This 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 [4] to the following:

**Definition IV.4.** An API misuse is the violation of one or more
API usage constraints. Such violation leads to misbehavior
of the API, e.g., errors, crashes, or vulnerabilities.

An example of an API misuse is not releasing a recently
opened resource by calling `close()` after the end of using
`java.io.FileReader` [57] (cf. section II).

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 defini-
tions 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 part of domain-specific
knowledge that is taken for granted by the API (designer).

V. CLASSIFICATION OF API USAGE CONSTRAINTS – FUM

We next provide **FUM**, a comprehensive „Framework for
API Usage constraint and Misuse classification“. To accurately
assess and understand the causes and types of API misuse,
one needs to elaborate on the different types of API usage
constraints. We consider an API usage constraint to be a
concrete constraint imposed by the API. Accordingly, an API
usage constraint type (FUM type) is a set of API usage
constraints that share the same kind of constraints.

Following our definitions (cf. definition IV.3 and IV.4) the
only cause of an API misuse is a violation of the API usage
constraint(s). Therefore, **FUM** is based on API usage
constraint types instead of API misuse types like other approaches
[2], [42]. Moreover, we localize each API usage constraint type
by the parts of an API method call (i.e., the API usage).

**Usage Constraint Types by Parts of an API Method Call**

As our study revealed, Monperrus et al. [47] so far intro-
duced the most comprehensive and fine-granular collection of
API directive kinds and an empirical study on their preval-
ence in API documentation, we extend their contribution.
We consider all their mentioned API directive kinds which
actually are API usage constraint types (cf. definition IV.3).
We then enrich their findings by contributions of previous
other studies [2], [6], [7], [10], [13], [54], [42], [43], [54],
[71], [72], [76], [80]. As a result, we introduce six new or
more fine-granular types to the API usage constraint types, i.e.,
**High-Level Constraints, Post-Null-Check, Controlling Method
Call, Threading, Argument State, and finally, Pre-Null-Check**
(annotated with “*” in Figure 2).

Since interaction with APIs is due to method invocations
[80], we assign each API usage constraint type to the different
parts of an API method call. We provide an overview of
API usage constraint types and their localization in Figure
2 Furthermore, we 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) which 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 return value, the method call itself,
and the passed arguments. There are also API usage constraint
types associated with multiple parts of the API method call
(uncolored dashed boxes).

A. FUM types associated with the “Return Value”

Here, we discuss the usage constraint types associated with
the return value of an API method, we call this main type
return value. 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 return
value (cf. green boxes in Figure 2).

**Post-Call(s)** constraints mandate calling methods on the re-
turn value [44], [47], [54]. For instance, the static method
AlgorithmParameters.getInstance(algorithm) [55] requires ini-
tializing the return value via the call of `init()` [47]. 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 return
value may be typed differently. In total, 0.9% of all analyzed
API elements contain such usage constraint [47].

**Post-Null-Check** comprises (implicit) usage constraints that
require a null check on a return value to avoid runtime errors
[2], [42], [54]. We can observe such constraints whenever a
method returns either a specified value or null. For instance,
the API class `java.io.InputStreamReader` [59] 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. [47]. Note that this kind
of usage constraint is distinguishable to `Post Calls(s)` because
no method call is used to check for null.

B. FUM types associated with the “Method Call”

We introduce all usage constraint types associated with
the method call itself (cf. red boxes in Figure 2) as method call.
Some APIs restrict interactions with themselves, 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.

Overall, 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 (cf. section V-D). Secondly, the state of the API object dedicates the allowed, required, or forbidden usage of itself [47].

**Method Call Sequence** are usage constraints enforcing the requirement to call methods in a particular sequence [10], [37]. A method call sequence can be considered as a usage protocol [6], [13], [71], [76]. Thus, the correct usage of method calls can be modeled with a finite-state machine [7], [71], [80]. Furthermore, the term typestate is also often used in the literature, i.e., the typestate defines the permitted operations on the API object based on its state [39]. In total, up to 12.2% of all analyzed API elements contain such usage constraint [47].

**Controlling Method Call** classifies constraints on a method call that is only allowed to be called if a condition is met beforehand [2], [6], [7], [34], [54]. Unlike Method Call Sequence, this usage constraint type aims to use method calls in conditional statements. Thus, 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 [60]. 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 API object that allows the call of next(). Only if calling 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. In fact, this is an extension of Method Call Sequence, as it can be seen as a sequence of method calls, however, 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. [47].

**Forbidden Method Call** specifies that certain methods are not allowed to be called [2], [47] and mostly is a special case of Method Call Sequence. There exist API methods that may not be called in an API object’s whole lifetime, e.g., deprecated methods which are still callable for backward compatibility [43]. Consider, for example, the static class java.net.URLDecoder [67] where the method decode(String) is deprecated, instead, an encoding scheme should be passed as second parameter of decode(String, String). Note, the most similar type of Monperrus et al. [47] is Method Call Visibility Directive addressing constraints on the calling context’s method call. Hence, we can only estimate the prevalence of Forbidden Method Call constraints with 4.2% [47] or less.

**C. FUM types associated with “Passed Arguments”**

We introduce usage constraint types related to passed arguments of an API method call (orange-colored box in Figure 3) as passed arguments.

**Argument State** constraints require the passed argument objects 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 [71], [76]. We can typically observe this type of usage constraint in the context of cryptography where several API objects need to be correctly composed [39]. Besides, we can observe special cases not perfectly fitting into this definition, e.g., 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. Since we tightly refer to Monperrus et al., we only distinguish between those. In fact, those special cases can be extended to other primitive types as well.

**String Format** constraints specify a restriction for the format of the passed string [47]. 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 [65] needs the argument 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 [47].

**Number Range** constraints specify that only a set or range of numbers are allowed to be passed [47], [72]. In total, 2.7% of
all analyzed API elements contain such usage constraint [47].

**Pre-Null-Check** constraints state the requirement to check for `null` before the argument is passed to the method [2], [42], [47], [54], [72] and are rather implicit like **Post-Null-Check** (cf. section V-A). There are cases where the documentation explicitly restricts to pass `null`, e.g., the `CollatingIterator` [5] 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 [47].

**Method Parameter Type** constraints restrict the passed argument’s type [47], [72]. Although this is contrary to object-oriented programming principles (i.e., generalization [77]), there are cases where this particular usage constraint exists [47]. For instance, the `java.sql.Timestamp` [66] API class offers the method `compareTo(date)` where the parameter `date` is from type `java.util.Date` [56]. However, the documentation states “Compares this Timestamp object to the given Date, which must be a Timestamp object” [47]. In total, 1.9% of all analyzed API elements contain such a usage constraint [47].

**Method Parameter Correlation** restricts the value of passed arguments because of their inter-dependency [47]. For instance, the method `setKeyEntry(alias, key, password, chain)` provided by `java.security.KeyStore` [63] requires passing a `key` depending on the provided certificate `chain`. In total, 1.9% of all analyzed API elements contain such usage constraints [47].

### D. FUM types with multiple API method call associations

In addition to API usage constraint types only assigned to one specific part of an API method call, there are also usage constraint types assigned to multiple parts we furthermore introduce, namely, **Exception Handling**, **Context**, and **High-Level Constraints** (uncolored and light-gray dashed boxes in Figure 2). Even though this category sounds similar to the multi-object variant of the sequential pattern introduced by Robillard et al. [71], their perspective is from that of an API misuse detector, and ours is from the API usage constraints themselves. Furthermore, our classification here involves more fine-grained information like exceptions, context, threading, etc. 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. **Exception Handling** constraints describe the situations in which possibly thrown exceptions must be considered and reacted to precisely [2], [34], [42]. 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). In total, 4.1% of all analyzed API elements contain such a usage constraint [47].

**Context** constraints are related to language constructs that must surround the API usage, excluding constructs that are already covered by **Exception Handling** and **Controlling Method Call**. We observed a similar usage constraint type mentioned in previous studies [2], [13], [47], but without explicitly limiting the constraint to surrounding language constructs. The `java.lang.Object` [64] is a good example that requires using its exposed method `wait()` only within loops. The most similar API directive mentioned by Monperrus et al. [47] is again the **Method Call Visibility Directive** we already introduced in the **Forbidden Method Call** constraint type (cf. section V-B). Similarly, we can only estimate the prevalence as 4.2% [47] or less. We further subdivide **Context** into two more usage constraint types, namely, **Synchronization** and **Synchronization** (cf. gray-colored box in Figure 2).

**Threading** addresses usage constraints of using an API only on specific threads. Consider, e.g., the `java.lang.Thread` [62] API class, where documentation states: “[all] Swing components and related classes […] must be accessed on the event dispatching thread” [62]. The most similar API directive mentioned by Monperrus et al. [47] is again the **Method Call Visibility Directive** as for **Forbidden Method Call** and **Context** with an estimated accumulated prevalence of 4.2% [47] or less.

**Synchronization** comprises requirements taking care of concurrent access to the API object in a multi-threaded environment (i.e., thread-safety) [2], [34], [47]. For instance, if at least one thread structurally modifies an instantiation of `java.util.HashMap` [58], there is a need to synchronize the API object externally (e.g., by retrieving a lock). This type also includes requirements for proper assurance of thread-safe use, e.g., a usage constraint of this type is violated if the lock is obtained twice in a nested manner (deadlock). In total, 3.9% of all analyzed API elements contain such a usage constraint [47].

**High-Level Constraints** address usage constraints that cannot be covered by the previously presented types in this section. They can cover all areas of an API usage (cf. large dashed box in Figure 2). For example, the operating system on which the program is executed on may be important — Windows machines provide different algorithms for the secure generation of random numbers than Linux machines [39]. Therefore, this may also affect the interaction with the given API that encapsulates and provide such functionality. Note that none of the studies we considered explicitly mentioned this type of usage constraint, neither did Monperrus et al. [47].

### E. FUM — Limitations

The work of Monperrus et al. [47] and Bruch et al. [9] 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, **FUM** (cf. Figure 2 in section V) only comprises usage constraint types that restrict an API’s usage (i.e., not an extension or inheritance) for simplicity. Furthermore, our survey and therefore **FUM** is based on the Java language (cf. section I).

### F. Overlapping Characteristics of FUM types

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,
VI. Discussing Differences of Classifications

The most similar classification framework compared to FUM is MuC by Amann [2] and its refined version provided by Li [42] (cf. section III.C). 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. [47], who examined the different types of API directives, including API usage constraints. Since Li used the work of Amann as foundation, they also implicitly used the findings from Monperrus et al. as well as we did for FUM. Although FUM works on the level of API usage constraint types, we can compare FUM to theirs as a violation of an API usage constraint as an API misuse (cf. definition IV.4), hence FUM also can be used to classify API misuses analogously. In this section, we show that FUM is at least as expressive as theirs. Moreover, FUM 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 “FUM” respectively.

a) API-Misuse Classification (MuC) [2]: MuC 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 FUM types or (ii) this subcategory does not refer to API misuse based on our definitions.

The category Method CallsFUM is covered by the violation of Method Call SequenceFUM as its constraints define the allowed and required method calls based on the API object’s state.

Their category ConditionsFUM is covered by the violation of a combination of FUM subcategories, namely Pre-Null-CheckFUM, Post-Null-CheckFUM, constraints of type Method Call SequenceFUM, and Controlling Method CallFUM as well as all the subtypes listed in Passed ArgumentsFUM (excluding Pre-Null-CheckFUM) and Argument TypeFUM. Thus, our separation allows us to assess API misuses more fine-grained. Further, Redundant Value and State ConditionFUM aims at the use of unnecessary conditional checks before using an API method. However, according to definition IV.4, they are not considered as an API misuse.

The subcategory Missing Synchronization ConditionsFUM 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 SynchronizationFUM usage constraints. In contrast, Redundant Synchronization ConditionsFUM classifies API misuses for two typical cases. First, situations in which a lock is obtained unnecessarily and secondly, 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 program (cf. definition IV.4). The second case is covered by the violation of SynchronizationFUM constraints.

Finally, their subcategory Missing Context ConditionsFUM aims at API misuses related to threading. For example, GUI components from SWING [62] 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 ThreadingFUM constraints. Moreover, we provide a more fine-granular view as our FUM type ContextFUM also comprises constraints related to language constructs that need to surround the API usage. The other subcategory Redundant Context ConditionsFUM classifies API misuses related to threading whose usages are merely redundant and therefore not needed which is covered in FUM as the violation of ThreadingFUM.

The category IterationFUM is covered by the violation of Controlling Method CallFUM and finally, the Exception HandlingFUM categories are covered by the violation of either Exception HandlingFUM or Method Call SequenceFUM.

b) Refinement of MuC by Li [42]: Following the study by Amann [2], Li [42] did a more exhaustive and wide-ranging empirical study on API misuses in the wild in 2020. They restructured and refined Amann’s MuC and introduced new subtypes in addition to Method CallsFUM, namely, Replaced ArgumentsFUM, Replaced NameFUM, Replaced Name and ArgumentsFUM, and finally, Replaced ReceiverFUM. FUM covers them with Method CallsFUM and Passed ArgumentsFUM.

Summary: FUM provides at least as much expressiveness as MuC [2] and Li’s [42] refined framework. Moreover, we provide a more natural view on API misuses by classifying usage constraint types instead of classifying API misuse types. We also provide a more fine-grained overview of usage constraints. FUM includes the (new) API usage constraint types Method Parameter TypeFUM, Method Parameter CorrelationFUM, and High-Level ConstraintFUM. In addition, FUM has split individual types to better locate them based on API method calls. For example, Missing Null CheckFUM is further distinguished in Post-Null-CheckFUM and Pre-Null-CheckFUM, which comes closer to the actual use of an API method call. As a result, this answers our second research question RQ2 for a classification framework.

VII. Case Study: CogniCrypt

To assess the extent to which FUM aids in determining and guiding the improvement of an API misuses detectors’ capabilities, we performed a case study consisting of two parts: First, we evaluate a domain-specific static analysis tool that, due to the domain-specificity, we expect to provide only partial coverage of FUM types. Second, we seek to assess what
The coverage of the domain-specific language CrySL \cite{38}, we sought to determine which API usage constraints (\textit{FUM} types) CogniCrypt can detect, and based on the assessment of the coverage, we used \textit{FUM} to identify weaknesses and to propose extensions to improve detection coverage. Specifically, we inspected which \textit{FUM} types are specifiable with CogniCrypt \cite{14} (version 1.0.0) and CrySL \cite{14} (version 2.0.0).

\textit{FUM} enabled us to classify the \textit{FUM} types of code samples of API misuses we collected from the \textit{MUBench Dataset} \cite{3}, as well as Li’s \cite{32} five most common API misuse patterns for each of their categories. In total, we were able to extract 88 usage constraint examples that are also API misuses based on definition \textit{IV}.4 (cf. section \textit{IV-D}). Our data set consists of general API misuse cases as we assess a domain-specific tool’s capabilities and then provide targeted improvements. The distribution of the examples for individual \textit{FUM} types is shown in Table \ref{table:summary} and the protocols of our evaluation per code sample can be found in the artifact \cite{74} we provide.

We examined whether the usage constraints could be specified via CrySL-rules. Upon inspection, only 27 of 88 samples were specifiable using CrySL. Due to bugs in CrySL and CogniCrypt we uncovered and reported \cite{74}, fewer code samples were detectable with our rule specifications. Table \ref{table:summary} shows the simplified results where we only discuss whether code samples of API usage constraints were specifiable with CrySL. Overall, the coverage of \textit{FUM} types (cf. column \textit{Currently}) is limited. However, this was to be expected since CrySL and CogniCrypt are designed specifically for the domain of cryptographic APIs (cf. definition \textit{IV.1} domain-specific API) and the scope of \textit{FUM} reaches far beyond just this domain.

As the next step of our case study, we assume that it is a sensible goal to adapt the domain of CrySL and CogniCrypt to general API misuse detection. Whether such a broadened domain makes sense with regards to usability or performance, is not focus of this evaluation. To see to which extent one could mitigate the identified weaknesses of CrySL and CogniCrypt, we derive theoretical improvements (in the form of language extensions) to CrySL detailed in the provided artifact \cite{74}. We focused on those \textit{FUM} types that had the highest prevalence in our test samples but which were not already fully covered. However, our derived code samples did not contain a sufficient amount of examples to derive a proper improvement suggestion for the remaining types of Multiple Assignments and Post-Call(s).

It seems plausible that, when implementing the suggested extensions, the number of specifiable samples would increase from 27 to 82 out of 88 (92\% in total). We discussed our findings with the developers of CrySL and CogniCrypt, which they will take into consideration for future development.

The case study revealed that \textit{FUM} can be practically used to assess the current development state of API misuse detectors and that further \textit{FUM} can be used to identify weaknesses and derive useful improvement suggestions.

<table>
<thead>
<tr>
<th>API Usage Constraints</th>
<th>Currently</th>
<th>Improvements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Return Value</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Passed Value</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>Passed Arguments</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Passed Arguments</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Passed Arguments</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Passed Arguments</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Passed Arguments</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>Method Call</td>
<td>19</td>
<td>19</td>
</tr>
<tr>
<td>Method Call</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>Method Call</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Multiple Assignments</td>
<td>14</td>
<td>14</td>
</tr>
<tr>
<td>Multiple Assignments</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Multiple Assignments</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>Multiple Assignments</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Multiple Assignments</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table I: Simplified results of the evaluation of CrySL’s coverage of \textit{FUM} types contrasted with proposed improvements.

\textbf{Current} shows CrySL’s coverage. \textbf{Improvements} shows the coverage once proposed improvements are applied. Dots symbolize the feasibility of specifying \textit{FUM} types and respectively detecting their violations, i.e., API misuses. \(\bullet\) \(\equiv\) no API usage constraints are specifiable for this type, \(\circ\) \(\equiv\) all API usage constraints are specifiable. The values show the number of examples per \textit{FUM} type.

\section{VIII. Conclusion}

This paper has presented three main contributions: (i) an overview of research on API misuses for the language Java, (ii) unified definitions around the term API misuse, and (iii) \textit{FUM} - a framework for API usage constraint and misuse classification which is based on the localization of usage constraints on a method-call level.

While classification frameworks already exist specifically for evaluating the capabilities of API misuse detectors (e.g., \textit{MuC} \cite{4}), \textit{FUM} focuses on an earlier level — API usage constraints instead of API misuses — because only the violation of API usage constraints are API misuses (cf. definition \textit{IV.4}).

We have shown that, compared to \textit{MuC}, \textit{FUM} has three advantages: (1) \textit{FUM} is more fine-granular with new API usage constraint types, (2) \textit{FUM} 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), and (3) \textit{FUM} was designed to be comprehensible and exhaustive for API designers/experts providing a more intuitive view on API misuses.

Furthermore, a case study with the API misuse detector CogniCrypt showed that \textit{FUM} can be practically used to assess the current development state of such detectors and that \textit{FUM} can effectively assist the identification of weaknesses and guide useful improvements.