A Systematic Evaluation of Static API-Misuse Detectors

Sven Amann, Hoan Anh Nguyen, Sarah Nadi, Tien N. Nguyen, and Mira Mezini, *Members, IEEE*

Abstract—Application Programming Interfaces (APIs) often have usage constraints, such as restrictions on call order or call conditions. *API misuses*, i.e., violations of these constraints, may lead to software crashes, bugs, and vulnerabilities. Though researchers developed many API-misuse detectors over the last two decades, recent studies show that API misuses are still prevalent. Therefore, we need to understand the capabilities and limitations of existing detectors in order to advance the state of the art. In this paper, we present the first-ever qualitative and quantitative evaluation that compares static API-misuse detectors along the same dimensions, and with original author validation. To accomplish this, we develop MuC, a classification of API misuses, and MuBenchPipe, an automated benchmark for detector comparison, on top of our misuse dataset, MuBench. Our results show that the capabilities of existing detectors vary greatly and that existing detectors, though capable of detecting misuses, suffer from extremely low precision and recall. A systematic root-cause analysis reveals that, most importantly, detectors need to go beyond the naive assumption that a deviation from the most-frequent usage corresponds to a misuse and need to obtain additional usage examples to train their models. We present possible directions towards more-powerful API-misuse detectors.

Index Terms—API-Misuse Detection, Survey, Misuse Classification, Benchmark, MUBench

arXiv:1712.00242v3 [cs.SE] 13 Mar 2018

1 Introduction

Incorrect usages of an Application Programming Interface (API), or *API misuses*, are violations of (implicit) *usage constraints* of the API. An example of a usage constraint is having to check that `hasNext()` returns `true` before calling `next()` on an `Iterator`, in order to avoid a `NoSuchElementException` at runtime. Incorrect usage of APIs is a prevalent cause of software bugs, crashes, and vulnerabilities [1]–[7]. While high-quality documentation of an API's usage constraints could help, it is often insufficient, at least in its current form, to solve the problem [8]. For example, a recent empirical study shows that Android developers prefer informal references, such as StackOverflow, over official API documentation, even though the former promotes many insecure API usages [9]. We confirm this tendency for a non-security API as well: Instances of `Iterator` may not be used after the underlying collection was modified, otherwise they throw a `ConcurrentModificationException`. Even though this constraint and the consequences of its violation are thoroughly documented, a review of the top-5% of 2,854 threads about `ConcurrentModificationException` on StackOverflow shows that 57% of them ask for a fix of the above misuse [10].

Ideally, development environments should assist developers in implementing correct usages and in finding and fixing existing misuses. In this paper, we focus on tools that identify misuses in a given codebase, specifically, those that automatically infer API-usage specifications and identify respective violations through static code analysis. We refer to these tools as *static API-misuse detectors*.

There have been many attempts to address the problem of API misuse. Existing static misuse detectors commonly mine *usage patterns*, i.e., equivalent API usages that occur frequently, and report any anomaly with respect to these patterns as potential misuse [1], [11]–[20]. The approaches differ in how they encode usages and frequency, as well as in the techniques they apply to identify patterns and violations thereof. Despite the vast amount of work on API-misuse detection, API misuses still exist in practice, as recent studies show [9], [21]. To advance the state of the art in API-misuse detection, we need to understand how existing approaches compare to each other, and what their current limitations are. This would allow researchers to improve API-misuse detectors by enhancing current strengths and overcoming weaknesses.

In this work, we propose the *API-Misuse Classification* (MuC) as a taxonomy for API misuses and a framework to assess the capabilities of static API-misuse detectors. In order to create such a taxonomy, we need a diverse sample of API misuses. In our previous work, we described MuBench, a dataset of 90 API misuses that we collected by reviewing over 1200 reports from existing bug datasets and conducting a developer survey [3]. MuBench provided us with the misuse examples needed to create a taxonomy. To cover the entire problem space of API misuses, for this paper, we add further misuses to this dataset by looking at examples from studies on API-usage directives [8], [22]. Using MuC, we qualitatively compare 12 existing detectors and identify their shortcomings. For example, we find that only few detectors detect misuses related to conditions or exception handling. We confirm this assessment with the detectors' original authors.

The previous step provides us with a conceptual comparison of existing detectors. We also want to compare these API-misuse detectors empirically, by both their precision and recall. This is a challenging task, due to the different underlying mechanisms

- *S. Amann and M. Mezini are with Technische Universität Darmstadt, Germany.*
  E-mails: amann@st.informatik.tu-darmstadt.de, mezini@informatik.tu-darmstadt.de
- *H. A. Nguyen is with Iowa State University, Iowa, United States of America.*
  E-mail: hoan@iastate.edu
- *S. Nadi is with University of Alberta, Canada.*
  E-mail: nadi@ualberta.ca
- *T. N. Nguyen is with University of Texas-Dallas, Texas, United States of America.*
  E-mail: tien.n.nguyen@utdallas.edu

Manuscript received July 1, 2017; revised January 20, 2018; accepted for publication March 12, 2018.

and representations used by detectors. To enable this empirical comparison, we build MuBenchPipe, the first automated pipeline to benchmark API-misuse detectors. Our automated benchmark leverages MuBench, and the additional misuses we collect in this work, and creates an infrastructure on top of it to run the detectors and compare their results. We perform three experiments based on 29 real-world projects and 25 hand-crafted examples to empirically evaluate and compare four state-of-the-art detectors. We exclude the other eight detectors since two rely on the discontinued Google Code Search [23], five target C/C++ code, and one targets Dalvik Bytecode, while our benchmark contains Java misuses. In Experiment P, we measure the precision of the detectors in a per-project setup, where they mine patterns and detect violations in individual projects from MuBench. In Experiment RUB, we determine upper bounds to the recall of the detectors with respect to the known misuses in MuBench. We take the possibility of insufficient training data out of the equation, by providing the detectors with crafted examples of correct usages for them to mine required patterns. Finally, in Experiment R, we measure the recall of the detectors against both the MuBench dataset and the detectors' own confirmed findings from Experiment P using a per-project setup.

Our conceptual analysis shows many previously neglected aspects of API misuse, such as incorrect exception handling and redundant calls. Our quantitative results show that misuse detectors are capable of detecting misuses, when provided with correct usages for pattern mining. However, they suffer from extremely low precision and recall in a realistic setting. We identify four root causes for false negatives and seven root causes for false positives. Most importantly, to improve precision, detectors need to go beyond the naive assumption that a deviation from the most-frequent usage corresponds to a misuse, for example, by building probabilistic models to reason about the likelihood of usages in their respective context. To improve recall, detectors need to obtain more correct usage examples, possibly from different sources, and to consider program semantics, such as type hierarchies and implicit dependencies between API usages. These novel insights are made possible by our automated benchmark. Our empirical results present a wake-up call, unveiling serious practical limitations of tools and evaluation strategies from the field. Foremost, detectors suffer from extremely low recall—which is typically not evaluated. Moreover, we find that the application of detectors to individual projects does not seem to give them sufficient data to learn good models of correct API usage.

In summary, this paper makes the following contributions to the area of API-misuse detection:

- A taxonomy of API misuses, MuC, which provides a conceptual framework to compare the capabilities of API-misuse detectors.
- A survey and qualitative assessment of 12 state-of-the-art misuse detectors, based on MuC.
- A publicly available automated benchmark pipeline for API-misuse detectors, MuBenchPipe, which facilitates systematic and reproducible evaluations of misuse detectors.
- An empirical comparison of both recall and precision of four existing misuse detectors using MuBenchPipe. Our work is the first to compare different detectors on both a conceptual and practical level and, more importantly, the first to measure the recall of detectors, unveiling their poor performance.
- A systematic analysis of the root causes for low precision and recall across detectors, to call researchers to action.

Our benchmarking infrastructure is publicly available [24] and our artifact Web page [10] provides full details on our results.

2 Background and Terminology

An *API usage* (*usage*, for short) is a piece of code that uses a given API to accomplish some task. It is a combination of basic *program elements*, such as method calls, exception handling, or arithmetic operations. The combination of such elements in an API usage is subject to constraints, which depend on the nature of the API. We call such constraints *usage constraints*. For example, two methods may need to be called in a specific order, division may not be used with a divisor of zero, and a file resource needs to be released along all execution paths. When a usage violates one or more such constraints, we call it a *misuse*, otherwise a *correct usage*.

The detection of API misuses may be approached through *static analyses* of source code or binaries and through *dynamic analyses*, i.e., runtime monitoring or analysis of runtime data, such as traces or logs. In either case, the detection requires either specifications of correct API usage to find violations of or specifications of misuses to find instances of. Such specifications may be *crafted manually* by experts or *inferred automatically* by algorithms. Automatic specification inference (or *mining*) may, again, be approached both *statically*, e.g., based on code samples or documentation, and *dynamically*, e.g., based on traces or logs.

Since manually crafting and maintaining specifications is costly, in this work, we focus on automated detectors. We call such tools *API-misuse detectors*. In the literature, we find *static misuse detectors*, which statically mine specifications and detect misuses through static analysis, e.g., [1], [13], [15]; *dynamic misuse detectors*, which dynamically mine specifications and detect misuses through dynamic analysis, e.g., [25], [26]; and *hybrid misuse detectors*, which, for example, combine dynamic specification mining with static detection [27]. In this work, we focus on static API-misuse detectors.

Static API-misuse detection is often achieved through detecting *deviant code* [1], [11]–[20], [28]. The key idea is that mistakes violate constraints that the code should adhere to and that, given sufficiently many examples of correct usage, such violations appear as *anomalies*. We call a usage that appears frequently in programs a *pattern*. The identification of mistakes through the detection of deviant code assumes that patterns correspond to correct usages (specifications) and anomalies with respect to these patterns are, consequently, misuses. Such an approach can detect mistakes in the usage of popular libraries [1], [11], [13], [15], [19], [28].

In our previous work [3], we collected a dataset of Java API misuses by reviewing bug reports of 21 real-world projects and surveying developers about API misuses. We call this dataset MuBench. It contains 90 misuses, 73 misuses from the real-world projects and 17 from the survey (see Table 1, Row 1). For each real-world misuse, the dataset identifies the *project* where the misuse is, the *project version* that contains the misuse, and the commit that fixed the misuse. For the other misuses, MuBench provides hand-crafted misuse examples and their fixes.

3 The API-Misuse Classification (MuC)

In this section, we introduce the *API-Misuse Classification* (MuC), our taxonomy for API misuses. We derive MuC from the misuse examples in the MuBench dataset. In Section 4, we use MuC to qualitatively compare the capabilities of existing API-misuse

Table 1: Datasets used throughout this paper, with the number of hand-crafted misuses (#HM), the number of real-world projects (#P), project versions (#PV), and misuses (#RM), and the total number of misuses (#M). "n/a" denotes that the number is not relevant for the use of the dataset.

	Dataset	#HM	#P	#PV	#RM	#M
1	Original MUBENCH [3]	17	21	55	73	90
2	Extended MUBENCH	27	21	55	73	100
3	Experiment P	n/a	5	5	n/a	n/a
4	Experiment RUB	25	13	29	39	64
5	Experiment R	0	13	29	53	53

detectors. In Section 7, we use MUC to define our expectations on the detectors' performance. Before presenting the classification itself, we briefly discuss existing related classifications to motivate the need for MUC.

3.1 Motivation for MUC

IEEE has a standard for classifying software defects [29], which served as the basis for IBM's ORTHOGONAL DEFECT CLASSIFICATION (ODC) [30]. The ODC uses the defect type as one of the aspects from which to classify defects. The defect type is composed of a conceptual program element, such as a function, check, assignment, documentation, or algorithm, and a violation type, i.e., either *missing* or *incorrect*. More recently, Beller *et al.* [31] presented the GENERAL DEFECT CLASSIFICATION (GCD), a remote ODC-descendant, tailored to compare the capabilities of automated static-analysis tools. Both classifications capture the entire domain of all types of software defects. To compare the capabilities of API-misuse detectors, we need a more fine-grained differentiation of a subset of the categories in both of them.

Past work presented empirical studies and taxonomies of API-usage directives [8], [22]. Many of these directives can be thought of as usage constraints in our terminology and their violations, consequently, as misuses. Other directives, however, do not formulate constraints. Examples are directives that explicitly allow `null` to be passed as a parameter and directives that inform about alternative ways to achieve a behaviour (possibly with different trade-offs). Therefore, we cannot directly convert a taxonomy of usage directives into a taxonomy of misuses. Instead, to consider the directives that can be viewed as usage constraints, we extend MUBENCH [3] by hand-crafted examples of misuses violating them, which we derive from examples in the studies. This gives us 10 additional misuses, resulting in a total of 100 misuses that we use for MUC and our experiments (see Table 1, Row 2). For simplicity, we subsequently refer to this extended dataset as MUBENCH.

3.2 The Classification

We developed MUC using a variation of Grounded Theory [32]: Following our notion of API misuses as API usages with one or more violations of usage constraints, the first author of this work went through all the misuses in MUBENCH and came up with labels for the characteristics of the respective violations, until each misuse was tagged with at least on label. Subsequently, all authors iteratively revisited the labelled misuses to unify semantically equivalent labels and group related labels, until we had a consistent taxonomy. In the end, we had two dimensions whose intersection describes all violations in MUBENCH: the type of the involved API-usage element and the type of the violation. Consequently, we define a *violation* as a pair of a violation type and an API-usage element.

Table 2: The Misuse Classification (MUC), with the number of misuses with a particular violation in MUBENCH.

	Violation Type	
API-Usage Element	Missing	Redundant
Method Call	30	13
Condition	48	6
`null` Check	25	3
Value or State	21	2
Synchronization	1	1
Context	1	1
Iteration	1	1
Exception Handling	10	1

An *API-usage element* is a program element that appears in API usages. The following elements are involved in the misuses in MUBENCH: *method calls*, *conditions*, *iterations*, and *exception handling*. Note that we consider primitive operators, such as arithmetic operators, as methods. For conditions, we further distinguish *`null` checks*, *value or state conditions*, *synchronization conditions*, and *context conditions*, because of their distinct properties.

The *violation type* describes how a usage violates a given usage constraint with respect to a given usage element. In MUBENCH, we find two violation types: *missing* and *redundant*. Violations of the missing type come from constraints that mandate the presence of a usage element. They generally cause program errors. An example of such a violation is a "missing method call." Violations of the redundant type come from constraints that mandate the absence of a usage element or declare the presence of a usage element unnecessary. Note that in either case the repetition of an element may have undesired effects, such as errors or decreased performance. An example of such a redundant violation is a "redundant method call."

Table 2 shows a summary of MUC. The numbers in the cells show how many misuses in MUBENCH have a respective violation. Note that a single misuse may have multiple violations; thus, the individual cells in the table sum up to more than 100. The table shows that missing method calls, `null` checks, and value or state conditions are the most prevalent violations. Redundant calls and missing exception handling are less frequent, but still prevalent, while we have only few examples of the other violations.

We now discuss the different violation categories shown in Table 2, grouped by the API-usage element involved.

Method Calls

Method calls are the most prominent elements of API usages, as they are the primary means of communication between client code and the API.

One violation category is *missing method calls*, which occur if a usage does not call a certain method that is mandated by the API usage constraints. For example, if a usage does not call `validate()` on a `JFrame` after adding elements to it, which is required for the change to become visible.

The other case is *redundant method calls*, which occur if a usage calls a certain method that is restricted by the API usage constraints. For example, if a usage calls `remove()` on a list that is currently being iterated over, which causes an exception in the subsequent iteration.

Conditions

Client code often needs to ensure conditions for valid communication to an API, in order to adhere to the API's usage constraints.

There are often alternative ways to ensure such conditions. For example, to ensure that a collection is not empty one may check `isEmpty()`, check its `size()`, or add an element to it. Note that checks, in particular, are also a means for the client code to vary usages depending on program inputs.

One violation category is *missing conditions*, which occur if a usage does not ensure certain conditions that are mandated by the API usage constraints. One case is *missing null checks*, e.g., if a usage fails to ensure that a receiver or a parameter of a call is not `null`. Another case is *missing value or state conditions*, e.g., if a usage fails to ensure that a `Map` contains a certain key before using the key to access the `Map`. In multi-threaded environments, *missing synchronization conditions* may occur, e.g., if a usage does not obtain a lock before updating a `HashMap` that is accessed from multiple threads [22]. Finally, *missing context conditions* may also occur, e.g., if a usage fails to ensure that GUI components in SWING are updated on the Event Dispatching Thread (EDT) [8].

The other case is *redundant conditions*, where a condition prevents a necessary part of a usage, e.g., a method call, from being executed along certain execution paths or is simply redundant. One case is *redundant `null` checks*, e.g., if the usage checks nullness only after a method has been invoked on the respective object. Another case is *redundant value or state conditions*, e.g., if the usage checks `isEmpty` on a collection that's guaranteed to contain an element. In multi-thread environments, *redundant synchronization conditions* may occur, e.g., if the usage requests a lock that it already holds, which may cause a deadlock. Finally, *redundant context conditions* may also occur, e.g., if a JUNIT assertion is executed on another thread, where its failing cannot be captured by the JUNIT framework.

Iteration

Iteration is another means of interacting with APIs, used, in particular, with collections and IO streams. It takes the form of loops and recursive methods. Note that respective usage constraints are about (not) repeating (part of) a usage, rather than about the condition that controls the execution.

One violation category is *missing iterations*, which occur if a usage does not repeatedly check a condition that the API usage constraints mandate must be checked again after executing part of the usage. For example, the Java documentation states that a call to `wait()` on an object should always happen in a loop that checks the condition the code waits for, because `wait()` could return before the condition is satisfied, in which case the usage should continue to wait.

The other case is *redundant iterations*, which occur if part of a usage is reiterated that the API usage constraints mandate may be executed not more than once or that is simply redundant. For example, a `Cipher` instance might be reused in a loop to encrypt a collection of values, but its initialization through calling `init()` must happen exactly once, i.e., before the loop. Note that in this situation, the required call is present in the respective code exactly once, as required, but its inclusion in an iteration causes a violation.

Exception Handling

Exceptions are a way for APIs to communicate errors to client code. The handling of different errors often depends on the specific API.

One violation category is *missing exception handling*, which occurs if a usage does not take actions to recover from a possible error, as mandated by the API usage constraints. For example, when initializing a `Cipher` with an externally provided cryptographic key, one should handle `InvalidKeyException`. Another example is resources that need to be closed after use, also in case of an exception. Such guarantees are often implemented by a `finally` block, but also using the try-with-resources construct or even respective handling in multiple `catch` blocks.

The other case is *redundant exception handling*, which occurs if a usage intercepts exceptions that should not be caught or handled explicitly. For example, catching `Throwable` when executing a command in an application might suppress a `CancellationException`, preventing the user from cancelling the execution.

4 Conceptual Classification of Existing Misuse Detectors

To advance the state of the art of API-misuse detection, we need to understand the capabilities and short-comings of existing misuse detectors. To identify detectors, we started from the publications about API-misuse detection listed in a survey of automated API-property inference techniques by Robillard *et al.* [33]. For each publication, we looked at all publications they refer to as related work and all publication that cite them, according to the ACM Digital Library or the IEEE Xplore Digital Library. We recursively repeated this process, until we found no new detectors.

We use MUC to guide the comparison. We provide a *conceptual classification* of the *capabilities* of each detector with respect to MUC, summarized in Table 3. We use the published description and results of each detector to identify which of MUC categories they can, conceptually, detect. To reduce subjectivity, we confirmed our capability assessment and the detector descriptions with the respective authors, except for PR-MINER and COLIBRI/ML, whose authors did not respond. We also describe the strategies used to evaluate each detector and summarize those in Table 4.

PR-MINER is a misuse detector for C [11]. It encodes usages as the set of all function names called within the same function and then employs frequent-itemset mining to find patterns with a minimum support of 15 usages. Violations here are strict subsets of a pattern that occur at least ten times less frequently than the pattern. To prune false positives, PR-MINER applies inter-procedural analysis, i.e., for each occurrence of a violation, it checks whether the missing call occurs within a called method. This analysis follows the call path for up to 3 levels. The reported violations are ranked by the respective pattern's support. PR-MINER focuses on detecting missing method-calls. The evaluation applied PR-MINER to three target projects individually, thereby finding violations of project-specific patterns. The detector reported 1,601 findings (1,447, 147, and 7 on the individual projects). The authors reviewed the top-60 violations reported across all projects and found 18.1% true positives (26.7%, 10.0%, and 14.3% on the individual projects).

CHRONICLER is a misuse detector for C [34]. It mines frequent call-precedence relations from an inter-procedural control-flow graph. A relation is considered frequent, if it holds on at least 80% of all execution paths. Paths where such relations do not hold are reported as violations. CHRONICLER detects missing method calls. Since loops are unrolled exactly once, it cannot detect missing iterations. The evaluation applied CHRONICLER to five projects, thereby finding violations of project-specific patterns. The authors compared the identified protocols with the documented protocols for one API and discussed a few examples of actual bugs found by their tool.

Table 3: Capabilities of Surveyed API-Misuse Detectors. ● denotes the capability to detect a violation. ◐ denotes the capability to detect a violation under special conditions. ○ denotes the inability to detect a violation.

	Method Calls		Conditions					Ex. Handl.		Iteration	
Misuses Detector	Missing	Redundant	Missing `null`	Missing Val./State	Missing Sync.	Missing Context	Redundant	Missing	Redundant	Missing	Redundant
PR-MINER [11]	●	○	○	○	○	○	○	○	○	○	○
CHRONICLER [34]	●	○	○	○	○	○	○	○	○	○	○
COLIBRI/ML [12]	●	○	○	○	○	○	○	○	○	○	○
JADET [13]	●	○	○	○	○	○	○	○	○	◐	○
RGJ07 [14]	◐	○	●	●	○	○	○	○	○	○	○
ALATTIN [18]	◐	○	●	◐	○	○	○	○	○	○	○
AX09 [16]	◐	○	◐	◐	○	○	○	●	○	○	○
CAR-MINER [17]	◐	○	○	○	○	○	○	●	○	○	○
GROUMINER [15]	●	○	◐	◐	○	○	○	○	○	◐	○
DMMC [36]	●	○	○	○	○	○	○	○	○	○	○
TIKANGA [19]	●	○	○	○	○	○	○	○	○	◐	○
DROIDASSIST [20]	●	●	○	○	○	○	○	○	○	○	○

Table 4: Summary of Empirical Evaluations of Surveyed API-Misuse Detectors. For the evaluation setup, IP denotes that detectors mine on the individual target projects and CP that they mine cross-project.

Detector	# of Target Projects	Eval. Setup	# of Reviewed Findings	Precision (Range)
PR-MINER [11]	3	IP	Top 60	18.1% (10-27%)
CHRONICLER [34]	5	IP	example-based	
COLIBRI/ML [12]	5	IP	example-based	
JADET [13]	5	IP	Top 10/project	6.5% (0-13%)
JADET [37]	20	CP	Top 25% (50)	8.0% (0-100%)
RGJ07 [14]	1	IP	example-based	
ALATTIN [18]	6	CP	Top 10/project	29.5% (13-100%)
AX09 [16]	3	IP	All (292)	90.4% (50-94%)
CAR-MINER [17]	5	CP	Top 10/project	60.1% (41-82%)
GROUMINER [15]	9	IP	Top 10/project	5.4% (0-8%)
DMMC [36]	1	IP	All (19)	73.7%
DMMC [1]	3	IP	Top 30	56.7%
TIKANGA [19]	6	IP	Top 25% (121)	9.9% (0-33%)
DROIDASSIST [20]			not evaluated	

COLIBRI/ML is another misuse detector for C [12]. It re-implements PR-MINER using *Formal Concept Analysis* [35] to strengthen the theoretical foundation of the approach. Consequently, its capabilities are the same as PR-MINER's. The evaluation applied COLIBRI/ML to five target projects, thereby finding violations of project-specific patterns. While some detected violations are presented in the paper, no statistics on the quality of the detector's findings are reported.

JADET is a misuse detector for Java [13]. It uses COLIBRI/ML [12], but instead of only method names, it encodes method-call order and call receivers in usages. It builds a directed graph whose nodes represent method calls on a given object and whose edges represent control flows. From this graph, it derives a pair of calls for each call-order relationship. The sets of these pairs form the input to the mining, which identifies patterns, i.e., sets of pairs, with a minimum support of 20. A violation may miss at most 2 properties of the violated pattern and needs to occur at least ten times less frequently than the pattern. Detected violations are ranked by $u \times s/v$, where s is the violated pattern's support, v is the number of violations of the pattern, and u is a uniqueness factor of the pattern. JADET detects missing method calls. It may detect missing loops as a missing call-order relation from a method call in the loop header to itself. The evaluation applied JADET to five target projects, thereby finding violations of project-specific patterns. The authors reviewed the top-10 violations reported per project and found 6.5% true positives (0%, 0%, 7.7%, 10.5%, and 13.3% on the individual projects). Other findings were classified as code smells (6.5%) or hints (35.0%).

In a subsequent study, JADET was applied in a cross-project setting where it was applied to 6,097 projects at once, using a minimum pattern support of 200 [37]. The authors reviewed the top-25% findings from a random sample of 20 projects, a total of 50 findings, and found 8% true positives. Other findings were classified as code smells (14.0%).

RGJ07 is a misuse detector for C [14]. It encodes usages as sets of properties for each variable. Properties are comparisons to literals, argument positions in function calls, and assignments. For each call, it creates a group of the property sets of the call's arguments. To all groups for a particular function, it applies sequence mining to learn common sequences of control-flow properties and frequent-itemset mining to identify all common sets of all other property types. Subsequently, it identifies violations of the common property sequences and sets. RGJ07 is designed to detect missing conditions. From the properties it encodes, it can detect missing `null` checks and missing value or state conditions. Since patterns contain preceding calls on arguments, it may also detect missing calls, if the respective call shares an argument with another call in the pattern. The evaluation applied RGJ07 to a single project, thereby finding violations of project-specific patterns. The authors discussed several examples of actual bugs their approach detects, but reported no statistics on the detection performance.

ALATTIN is a misuse detector for Java [18], specialized in alternative patterns for condition checks. For each target method `m`, it queries a code-search engine to find example usages. From each example, it extracts a set of rules about pre- and post-condition checks on the receiver, the arguments, and the return value of `m`, e.g., "`boolean` check on `return` of `Iterator.hasNext` before `Iterator.next`." It then applies frequent-itemset mining on these rules to obtain patterns with a minimum support of 40%. For each such pattern, it extracts the subset of all groups that do not adhere to the pattern and repeats mining on that subset to obtain infrequent patterns with a minimum support of 20%. Finally, it combines all frequent and infrequent patterns for the same method by disjunction. An analyzed method has a violation if the set of rules that hold in it is not a superset of any of the alternative patterns. Violations are ranked by the support of the respective pattern. ALATTIN, therefore, detects missing `null`-checks and missing value or state conditions that are ensured by checks and do not involve literals. It may also detect missing method-calls that occur in checks. The evaluation applied ALATTIN to six projects. Since it queries code-search engines for usage examples, it detects violations of cross-project patterns. The authors manually reviewed all violations of the top-10 patterns per project, a total of 532 findings, and confirmed that 29.5% identify missing condition checks (12.5%, 26.2%, 28.1%, 32.7%, 52.6%, and 100% for the individual projects). Considering frequent alternative patterns reduced false positives by 15.2% on average, which increased precision to 33.3%. Considering both frequent and infrequent alternatives even reduced false positives by 28.1% on average, leading to a precision of 37.8%, but introduced 1.5% additional false negatives, because misuses that occur multiple times are mistaken for infrequent patterns.

AX09 is a misuse detector for C [16], specialized in detecting wrong error handling, realized through returning (and checking for) error codes. It distinguishes normal paths, i.e., execution paths from the beginning of the `main` function to its end, and error paths, i.e., paths from the beginning of the `main` function to an `exit` or `return` statement in an error-handling block. AX09 uses

push-down model checking to generate such paths as sequences of method calls and applies frequent-subsequence mining to find patterns with a minimum support of 80% (but at least 5 usages). It then uses push-down model checking to verify adherence to these patterns and identify respective violations. Finally, it filters false positives by tracking variable values and excluding error cases that cannot occur. It detects missing error-handling as well as missing method calls among error-handling functions. Since it identifies error-handling blocks through a predefined set of checks, it also detects missing `null`-checks and missing value or state conditions in the case of missing error-handling blocks. The evaluation applied AX09 to three projects individually, thereby finding violations of project-specific patterns. The authors manually reviewed all 292 findings and confirmed 90.4% true positives (50.0%, 90.3%, and 93.5% on the individual projects).

CAR-MINER is a misuse detector for C++ and Java [17], also specialized in detecting wrong error handling. For each analyzed method `m` in a given code corpus, it queries a code-search engine to find example usages. From the examples, it builds an *Exception Flow Graph* (EFG), i.e., a control-flow graph with additional edges for exceptional flow. From the EFG, it generates *normal* call sequences that lead to the currently analyzed call and *exception* call sequences that lead from the call along exceptional edges. Subsequently, it mines association rules between normal sequences and exception sequences, with a minimum support of 40%. To detect violations, CAR-MINER extracts the normal call sequence and the exception call sequence for the target method call. It then uses the learned association rules to determine the expected exception handling and reports a violation if the actual sequence does not include it. CAR-MINER detects missing exception-handling as well as missing method calls among error-handling functions. The evaluation applied CAR-MINER to five projects. Since it queries code-search engines for usage examples, it detects violations of cross-project patterns. The authors manually reviewed all violations of the top-10 association rules for each project, a total of 264 violations, and confirmed that 60.1% identify wrong error handling (41.1%, 54.5%, 68.2%, 68.4%, and 82.3% on the individual projects). Other findings were classified as hints (3.0%).

GROUMINER is a misuse detector for Java [15]. It creates a graph-based object-usage representation (GROUM) for each target method. A GROUM is a directed acyclic graph whose nodes represent method calls, branchings, and loops and whose edges encode control and data flows. GROUMINER performs frequent-subgraph mining on sets of such graphs to detect recurring usage patterns with a minimum support of 6. When at least 90% of all occurrences of a sub-pattern can be extended to a larger pattern, but some cannot, those *rare* inextensible occurrences are considered as violations. Note that such violations have always exactly one node less than a pattern. The detection of patterns and violations happens at the same time. Violations are ranked by their *rareness*, i.e., the support of the pattern over the support of the violation. GROUMINER detects missing method calls. It also detects missing conditions and loops at the granularity of a missing branching or loop node. However, it cannot consider the actual condition. The evaluation applied GROUMINER to nine projects individually, thereby finding violations of project-specific patterns. The authors reviewed the top-10 violations per project, a total of 184 findings, and found 5.4% true positives (three times 0%, five times 6.7%, and once 7.8% on the individual projects). Other findings were classified as code smells (7.6%) or hints (6.0%).

DMMC is a misuse detector for Java [36], specialized in missing method calls. The detection is based on type usages, i.e., sets of methods called on an instance of a given type in a given method. Two usages are *exactly similar* if their respective sets match and are *almost similar* if one of them contains exactly one additional method. The detection is based on the assumption that violations should have only few exactly-similar usages, but many almost-similar ones. The likelihood of a usage x being a violation is expressed in the *strangeness score* $= 1 - |E(x)|/(|E(x)| + |A(X)|)$, where $E(x)$ is the set of usages that are exactly similar to x and $A(x)$ the set of those that are almost similar. A usage is considered a violation if its strangeness score is above 0.97. Violations are ranked by the strangeness score. DMMC detects misuses with exactly one missing method-call. The evaluation applied DMMC to a single project, thereby finding project-specific violations. The authors manually reviewed all findings with a strangeness score above 97%, a total of 19 findings, and confirmed 73.7% as true positives. The evaluation was repeated later [1], applying DMMC to three projects individually, thereby finding project-specific violations for a predefined set of APIs. The authors report that they manually reviewed approximately 30 findings, and confirmed 17 ($\approx$ 56.7%) as true positives. Others were classified as workarounds for bugs inside a used API.

TIKANGA is a misuse detector for Java [19] that builds on JADET. It extends the simple call-order properties to general Computation Tree Logic formulae on object usages. Specifically, it uses formulae that require a certain call to occur, formulae that require two calls in order, and formulae that require a certain call to happen after another. It uses model checking to determine all those formulae with a minimum support of 20 in the codebase. Violations are ranked by the *conviction* measure [38] of the association between the set of present formulae and the set of missing formulae in the violating usage. It then applies Formal Concept Analysis [35] to obtain patterns and violations at the same time. TIKANGA's capabilities are the same as JADET's. The evaluation applied TIKANGA to six projects individually, finding violations of project-specific patterns. The authors manually reviewed the top-25% of findings per project, a total of 121 findings, and confirmed 9.9% as true positives (0%, 0%, 8.3%, 20.0%, 21.4%, and 33.3% on the individual projects). Other findings were classified as code smells (29.8%).

DROIDASSIST is a detector for Android Java Bytecode [20]. It generates method-call sequences from source code and learns a Hidden Markov Model from them, to compute the likelihood of a particular call sequence. If the likelihood is too small, the sequence is considered a violation. DROIDASSIST then explores different modifications of the sequence (adding, replacing, and removing calls) to find a slightly modified, more likely sequence. This allows it to detect missing and redundant method calls and even to suggest solutions for them. An evaluation of this mechanism is not provided in the respective paper.

Summary. All detectors use code (snippets) as training and verification input. Some require the code in a compiled format, such as Java Bytecode, while others directly work on source code. Detectors typically encode usages as sets, sequences, or graphs. Graph representations seem promising for simultaneously encoding usage elements, order, and data-flow relations. With the exception of DROIDASSIST and DMMC, detectors mine patterns through frequent-itemset/subsequence/subgraph mining, according to their usage representation. To detect violations, they mine in-extensible

parts of patterns that are themselves observed infrequently. This implies that they cannot detect redundant elements, since a usage with such an element is never part of any pattern. The exception is DROIDASSIST, which might find redundant calls as being unlikely.

Table 3 summarizes the detectors' capabilities with respect to MUC. Overall, we find that detectors cover only a small subset of all API-misuse categories. While all detectors may, to some degree, identify missing method calls, only four detectors may identify missing `null` checks and missing value-or-state conditions, only three may identify missing iterations, and only two may identify missing exception handling. None of the detectors targets all of these categories.

Existing detectors use both absolute and relative minimum support thresholds to identify patterns. The exceptions are, again, DROIDASSIST and DMMC, which use probabilistic approaches. Since many detectors produce a high number of false positives, they use a variety of ranking strategies. Most of these rely mainly on the pattern support, but some use different concepts, such as *rareness*, *strangeness*, or *conviction*. A comparison of different ranking strategies is not reported in any of the publications.

Table 4 summarizes the empirical evaluations of the surveyed detectors, as reported in their original papers. Most evaluations apply detectors to target projects individually. In this setting, the detectors learn project-specific patterns and identify respective violations The number of projects ranges from 1 to 20 (average 5.3; median 5). The concrete projects samples are all distinct and mostly even disjunct.

To assess the detection performance, most authors review the top-X findings of their detectors, where X is a fixed number or percentage. They then either present anecdotal evidence of true positives or measure the precision of detectors. Many evaluations also present additional categories of findings, such as code smells, to distinguish false positives from other non-misuse findings that may still be valuable to developers. The definitions of when a finding belongs to which category—if provided—differ between publications, even if they use the same label, e.g., "bug" or "code smell." No evaluation considers the recall of the respective detector.

Overall, it appears that the detectors that focus on specific violations, such as error handling or missing method calls, have higher precision. However, simply comparing detectors based on their reported empirical results would be unreliable, since the target projects, the review sample sizes, and the criteria to assess detector findings differ between the studies.

5 EXPERIMENTAL SETUP

In Section 4, we conceptually compared detectors' capabilities. In this section, we describe the experimental setup we use to *empirically* compare their capabilities. We design three experiments, to measure both the detectors' precision and recall. We build these experiments on MUBENCH as a ground-truth dataset. This enables us to compare all detectors on the same target projects and with respect to the same known misuses.

Subject Detectors. In this study, we focus on misuse detectors for Java APIs, because MUBENCH contains examples of Java-API misuses. Our survey identifies seven such detectors. We contacted the respective authors and got responses from all of them. However, we learned that we cannot run CAR-MINER and ALATTIN, because they both depend on Google Code Search, a service that is no longer available [23]. We exclude DROIDASSIST, because its implementation only supports Dalvik Bytecode,[1] while the examples in MUBENCH are general Java projects, which compile to Java Bytecode. This leaves us with four detectors JADET, GROUMINER, TIKANGA, and DMMC.

Misuse Dataset. We use MUBENCH, described in Section 2, to find targets for our evaluations. While GROUMINER works on source code, JADET, TIKANGA, and DMMC require Java Bytecode as input. Thus, we can only compare them on project versions for which we have both source code and Bytecode. Since Bytecode is not readily available for most project versions in the dataset, we resort to compiling them ourselves by adding necessary build files and fixing any dependency issues. We exclude 26 project versions (47%) with compilation errors that we could not fix. In the end, we have 29 compilable project versions and 25 hand-crafted examples, with 64 misuses in total, for our experiments. Note that some project versions contain multiple misuses. The last three rows in Table 1 describe the subsets of this dataset that we use in the individual experiments. We publish the dataset [24] for others to use in future studies.

5.1 Experiment P

We design Experiment P to assess the precision of detectors.

Motivation. Past studies show that developers rarely use analysis tools that produce many false positives [39]–[41]. Therefore, for a detector to be adopted in practice, it needs a high precision.

Setup. To measure precision, we follow the most-common experimental setup we found in the literature (cf. Table 4). First, we run detectors on individual project versions. In this setting, they mine patterns and detect violations on a per-project basis. Second, we manually validate the top-20 findings per detector on each version, as determined by the respective detector's ranking strategies. We limit the number of findings, because it seems likely that developers would only consider a fixed number of findings, rather than all of a potentially very large number of findings. Hence, the precision in a detector's top findings is likely crucial for tool adoption. Also, we need to limit the effort of reviewing findings of multiple detectors on each project version.

Dataset. Since manually reviewing findings of all detectors on all project versions is infeasible, we sample five project versions. To ensure a fair selection of projects, we first run all detectors on all project versions. For practical reasons, we timeout each detector on an individual project version after two hours. The run statistics are summarized in Table 5.

JADET and TIKANGA fail on one project version and DMMC fails on four project versions, since the Bytecode contains constructs that the detectors' respective Bytecode toolkits do not support. GROUMINER times out on eight project versions and produces an error on one other version. We exclude any project version where a detector fails.

For the remaining 15 versions, we observe that the total number of findings correlates across detectors. Table 6 shows that the pairwise correlation (Pearson's r) is strong (≥ 0.75) or medium (≥ 0.5) for all pairs of detectors, except for JADET and GROUMINER ($r = 0.49$). This means that either all detectors report a relatively large or a relatively small number of findings on any given project version. We hypothesise that the total number of findings might be related to the detectors' ability to precisely identify misuses

1. A bytecode format developed by Google, which is optimized for the characteristics of mobile operating systems (especially for the Android platform).

Table 5: Number of Findings per Detector on All Compilable Project Versions in MUBENCH. Experiment P includes the two projects with the highest number of findings, the two projects with the lowest number of findings, and one randomly selected project.

Project	Version	Number of Findings					Sample Criterion
		JADET	GROUMINER	TIKANGA	DMMC	Norm. Avg.	
APACHE COMMONS LANG	587	0	28	0	157	0.06	
APACHE COMMONS MATH	998	17	error	17	686	0.20	
ADEMPIERE	1312	0	27	0	116	0.05	
ALIBABA DUID	e10f28	17	timeout (2h)	5	520	0.13	
CLOSURE	114	113	101	24	1233	0.49	
CLOSURE	319	176	126	45	1945	0.74	highest
CLOSURE	884	71	167	33	1966	0.63	
APACHE HTTPCLIENT	302	0	12	0	114	0.03	
APACHE HTTPCLIENT	444	0	15	0	110	0.03	
APACHE HTTPCLIENT	452	0	12	0	113	0.03	
ITEXT	5091	17	198	55	1138	0.55	highest
APACHE JACKRABBIT	1601	12	186	22	error	0.41	
APACHE JACKRABBIT	1678	0	15	0	error	0.03	
APACHE JACKRABBIT	1694	13	186	22	error	0.41	
APACHE JACKRABBIT	1750	10	timeout (2h)	8	434	0.12	
JFREECHART	103	167	timeout (2h)	88	673	0.69	
JFREECHART	164	168	timeout (2h)	90	664	0.69	
JFREECHART	881	194	timeout (2h)	93	745	0.76	
JFREECHART	1025	194	timeout (2h)	93	747	0.76	
JFREECHART	2183	190	timeout (2h)	100	906	0.81	
JFREECHART	2266	195	timeout (2h)	102	913	0.82	
JMRTD	51	0	11	0	29	0.02	lowest
JMRTD	67	0	10	0	35	0.02	
JODA-TIME	1231	0	0	0	1	0.00	lowest
APACHE LUCENE	207	0	140	0	182	0.20	
APACHE LUCENE	754	0	54	0	265	0.10	
APACHE LUCENE	1251	2	62	0	error	0.11	
APACHE LUCENE	1918	2	88	4	583	0.20	random
MOZILLA RHINO	286251	error	55	error	257	0.20	

Table 6: Correction of the Number of Findings per Project Version For All Pairs of Detectors (Pearson's r). Strong correlation ($r \geq 0.75$) in **bold**. Medium correlation ($r \geq 0.5$) in *italic*.

	JADET	GROUMINER	DMMC	TIKANGA
JADET	**1.00**			
GROUMINER	0.49	**1.00**		
DMMC	**0.85**	**0.78**	**1.00**	
TIKANGA	*0.70*	**0.82**	**0.88**	**1.00**

in a given project version. Therefore, we sample project versions according to the average normalized number of findings across all detector. We normalize the number of findings per detector on all project versions by the maximum number of findings of that detector on any project version. We sample the two projects with the highest average normalized number of findings across all detectors (CLOSURE [42] v319 and ITEXT [43] v5091) and the two projects with the lowest average normalized number of findings across all detectors (JMRTD [44] v51 and JODA-TIME [45] v1231). Additionally, we randomly select one more project version (APACHE LUCENE [46] v1918) from the remaining projects, to cover the middle ground. Note that we select at most one version from each distinct project, because different versions of the same project may share a lot of code, such that detectors are likely to perform similarly on them. This dataset for Experiment P is summarized in Row 3 of Table 1.

Metrics. We calculate the precision of the detector, i.e., the ratio between the number of true positives over the number of findings.

Review Process. Two authors independently review each of the top-20 findings of the sampled project versions and mark it as a misuse or not. To determine this, they consider the logic and the documentation in the source code, the API's documentation, and its implementation if publicly available. After the review, any disagreements between the reviewers are discussed until a consensus is reached. We report Cohen's Kappa score as a measure of the reviewers' agreement. Note that we follow a lenient reviewing process. For example, assume a usage misses a check `if (iterator.hasNext())` before calling `iterator.next()`. If the detector finds that `hasNext()` is missing, we mark the finding as a hit, even though this does not explicitly state that the call to `next()` should be guarded by a check on the return value of `hasNext()`. This follows our intuition that such findings may still provide a developer with a valuable hint about the problem.

5.2 Experiment RUB

We design Experiment RUB to assess the detection capabilities of our subject detectors, i.e., to measure an upper bound to their recall under the assumption that they always mine the required pattern.

Motivation. We argue that it is important for developers to know which misuses a particular tool may or may not find, in order to decide whether the tool is adequate for their use case and whether they must take additional measures. Moreover, it is important for researchers to know which types of misuses existing detectors may identify, in order to direct future work. Therefore, we measure detectors' recall while providing sufficiently many correct usages that would allow them to mine the required pattern.

Dataset. For this experiment, we use all compilable project versions from the MUBENCH dataset with the respective known misuses, as well as the hand-crafted misuse examples. This dataset for Experiment RUB is summarized in Row 4 of Table 1.

Setup. Recall that all our subject detectors mine patterns, i.e., frequently reoccurring API usages, and assume that these correspond to correct usages. They use these patterns to identify misuses. Recall further that each detector has a distinct representation of usages and patterns and its own mining and detection strategies. If a detector fails to identify a particular misuse, this may be due to (1) an inherent limitation of the detector, e.g., because it cannot represent some usage element such as conditions, or (2) a lack of examples of respective correct usage for pattern mining, i.e., a limitation of the training data. With Experiment RUB, we focus on (1), i.e., we take (2) out of the equation and assess the detectors' general ability to identify misuses. To this end, we provide the detectors with sufficiently many examples of correct usage corresponding to the misuses in question. This guarantees that they could mine a respective pattern. If the detector is unable to identify a misuse in this setting, we know the problem lies with the detector itself.

We manually create a correct usage for each misuse in the dataset, using the fixing commits recorded in MUBENCH. For each misuse, we take the entire code of the method with the misuse after the fixing commit and remove all code that has no data or control dependencies to the objects involved in the misuse. We store the code of this *crafted correct usage* in our dataset.

In the experiment, we run each detector once for each individual known misuse in the dataset. In each run, we provide the detector with the file that contains the known misuse and with 50 copies of the respective crafted correct usage. We ensure that the detector considers each copy as a distinct usage. We configure the detectors to mine patterns with a minimum support of 50, thereby ensuring that they mine patterns only from the code in the crafted correct usage. We chose 50 as a threshold, since it is high enough to ensure that no detector mines patterns from the code in the file with the misuse.

Metrics. We calculate two numbers for each detector. The first is its *conceptual recall upper bound*, which is the fraction of the known misuses in the dataset that match its capabilities from Table 3. Note that the conceptual recall upper bound is calculated offline, without running any experiments. The second is the detector's *empirical recall upper bound*, which is the fraction of misuses a detector actually finds from all the known misuses in the dataset. An ideal detector should have an empirical recall upper bound equal to its conceptual recall upper bound. Otherwise, its practical capabilities do not match its conceptual capabilities. In such cases, we investigate the root causes for such mismatches. Note that we use the term "upper bound," because neither recall rate reflects the detectors' recall in a setting without guarantees on the number of correct usages for mining.

Review Process. To evaluate the results, we review all *potential hits*, i.e., findings from each detector that identify violations in the same files and methods as known misuses. Two authors independently review each such potential hit to determine whether it actually identifies one of the known misuses. If at least one potential hit identifies a misuse, we count it as a *hit*. After the review, any disagreements between the reviewers are discussed until a consensus is reached. We report Cohen's Kappa score as a measure of the reviewers' agreement. We follow the same lenient review process as for Experiment P.

5.3 Experiment R

We design Experiment R to assess the recall of detectors.

Motivation. While Experiment RUB gives us an upper bound to the recall of misuse detectors, we also want to assess their actual recall where we do not provide them with correct usages ourselves. Due to the lack of a ground-truth dataset, such an experiment has not been attempted before in any of the misuse-detection papers we surveyed.

Dataset. As the ground truth for this experiment, we use all known misuses from real-world projects in MUBENCH plus the true positives identified by any of the detectors in Experiment P. This means that Experiment R not only evaluates recall against the misuses of MUBENCH, but also practically cross-validates the detector capabilities against each other. We exclude the hand-crafted misuse examples from this experiment, since there is no corresponding code for the detectors to mine patterns from. The dataset we use for Experiment R is summarized in Row 5 of Table 1.

Setup. We run all detectors on all projects versions individually, i.e., we use the same per-project setup as for Experiment P.

Metrics. We calculate the recall of the detectors, i.e., the number of actual hits over the number of known misuses in the dataset.

Review Process. We review all potential hits in the same process as for Experiment RUB. This gives us the detectors' recall with respect to a large number of known misuses from MUBENCH.

6 MUBENCHPIPE

To systematically assess and compare API-misuse detectors, we built MUBENCHPIPE, a benchmarking pipeline for API-misuse detectors. MUBENCHPIPE automates large parts of the experimental setup presented in Section 5 and facilitates the reproduction of our study. It also enables adding new detectors to the comparison, as well as benchmarking with different or extended datasets, in the future. We publish the pipeline [24] for future studies.

6.1 Automation

Following the idea of automated bug-detection benchmarks for C programs, such as BUGBENCH [47] and BEGBUNCH [48], we facilitate the benchmarking of multiple detectors on our misuse dataset with an evaluation pipeline. MUBENCHPIPE automates many of our evaluation steps, such as retrieval and compilation of target projects, running detectors, and collecting their findings. MUBENCHPIPE provides a command-line interface to control these steps. We subsequently describe the pipeline steps we implemented to facilitate our evaluation.

Checkout. MUBENCHPIPE uses the recorded commit Id from MUBENCH to obtain the source code of the respective project version. It supports SVN and Git repositories, source archives (zip), as well as a special handling for the hand-crafted examples that come with MUBENCH.

Compile. For every project version, MUBENCHPIPE first copies the entire project source code, the individual files containing known misuses, and the respective crafted correct usages for Experiment RUB each into a separate folder. It then uses the respective build configuration from the dataset to compile all Java sources to Bytecode. After compilation, it copies the entire project Bytecode, the Bytecode of the individual files containing known misuses, and the Bytecode of the respective crafted correct usages each into a separate folder. This way, we may provide the detectors with the source code or Bytecode of each of these parts individually.

Detect. For each detector, we also built a *runner* to have a unified command-line interface for all detectors. For every project version,

MUBENCHPIPE invokes the detector with the paths to the respective source code and Bytecode. All detectors are invoked with the best configuration reported in their respective publication. Apart from adding some accessor methods that allow us to obtain the detectors' output, all detector implementations were left unchanged.

Validation. To help with the manual review of findings, MUBENCHPIPE automatically publishes experiment results to a review website [10]. For every detector finding, the website shows the source code it is found in along with any metadata the detector provides, such as the violated pattern, the properties of the violation, and the detector's confidence.

For Experiments RUB and R, MUBENCHPIPE automatically filters potential hits, by matching findings to known misuses by file and method name. On the review website, a reviewer sees the description of the known misuse as well as its fix, along with the set of potential hits that need to be reviewed. For Experiment P, MUBENCHPIPE shows all findings of the detector on the review site.

The review website allows reviewers to save an assessment and comment for each finding. It also ensures at least two reviews for each finding, before automatically computing the experiment statistics, such as precision, recall, and Cohen's Kappa scores.

6.2 Reproduction, Replication, and Extension

MUBENCHPIPE comes with a Docker image, which allows running reproducible experiments across platforms, without the need to ensure a proper environment setup. Its review website comes with a second Docker image, which allows serving it standalone. Moreover, it is based on PHP and MySQL, such that it can be hosted on any off-the-shelf webspace. The review website facilitates independent reviews, even when researchers work from different locations, while ensuring review integrity using authentication. The website may also directly be used as an artifact to publish review results and experiment statistics. MUBENCHPIPE defines a simple data schema for misuse examples to facilitate extensions of MUBENCH. It also provides a convenient Java interface as a Maven dependency to enable plugging in additional detectors for evaluation on the benchmark. For further details on how to use or extend MUBENCHPIPE, we refer the readers to our project website [24].

7 RESULTS

We now discuss the results of comparing JADET, GROUMINER, TIKANGA, and DMMC in our experiments. All reviewing data is available on our artifact page [10].

7.1 Experiment P

Table 7 shows our precision results, based on reviewing the top-20 findings per detector on each of our five sample projects. The second column shows the total number of reviewed findings, 230 in total across all detectors. Note that all detectors report less than 20 findings for some projects. The third column shows the confirmed misuses after resolving disagreements, and the fourth column shows the precision with respect to the reviewed findings. The fifth column shows the Kappa score for the manual reviews, and the remaining columns show the frequencies of root causes for false positives. We find that the precision of all detectors is extremely low. TIKANGA shows the best precision of only 11.4%. JADET and DMMC follow immediately behind, with a precision of 10.3% and 9.9%, respectively. GROUMINER reports only false positives in its top-20 findings.

> *O1:* All detectors have extremely low precision (below 12%). On average, they report less than 1.5 actual misuses in their top-20 findings.

The Kappa scores indicate high reviewer agreement, which shows that all detectors produced mostly clear false positives. The score is a little lower for TIKANGA, because it reported one confirmed misuse twice, which one of the reviewers first accepted as an actual hit while the other did not. The score is also lower for DMMC, because we initially disagreed on several violations it identifies in `Iterator` usages that do not check `hasNext()`, but the underlying collection's size.

True Positives

Out of the 230 reported findings we reviewed, we confirm 17 true misuses. DMMC reports 8 misuses of an iterator API where `hasNext()` is not checked. JADET reports 4 misuses that access a collection without checking its size before. Also for collections, TIKANGA reports 4 misuses with a missing `hasNext()` and 1 misuse with a missing size check. One misuse is reported by both TIKANGA and JADET and another by both TIKANGA and DMMC. Additionally, JADET reports one misuse twice. This leaves a total of 14 unique misuses, all different from the known misuses in MUBENCH. Interestingly, *all* these misuses are missing value or state conditions, for which the detectors report only missing calls to methods that should be used in the respective missing checks. We accept these findings in our lenient review process.

> *O2:* All 14 confirmed misuses in Experiment P are missing value or state condition checks before accessing the elements of a collection, either directly or through an iterator.

False Positives

To identify opportunities to improve the precision of misuse detectors, we systematically investigate the root causes for the false positives they report. In the following, we discuss these root causes summarized across all detectors, in the order of their absolute frequency.

1. Uncommon. Particular usages may violate the patterns that detectors learn from frequent usages, without violating actual API usage constraints. Detectors cannot differentiate infrequent from invalid usage. For example, DMMC and JADET learn that the methods `getKey()` and `getValue()` of `MapEntry` usually appear together in code. They both report violations if a call to either of these methods is missing, or, in case of JADET, if the calls appear in a different order. However, there is no requirement by the API to always call both getter methods, let alone in a specific order. Across the reported violations we analyzed, the detectors falsely report 42 missing method calls in cases where one out of a number of getter methods is missing or invoked in a different order. Another example is that JADET and TIKANGA learn that methods such as `List.add()` and `Map.put()` are usually invoked in loops and report five missing iterations for respective invocations outside a loop, which are perfectly fine according to the API. Approaches such as multi-level patterns [49] or ALATTIN's alternative patterns [18] may help to mitigate this problem. Also note that the four detectors in our experiments all use absolute frequency thresholds, while some of the detectors from our survey

Table 7: Experiment P: Precision of the Detectors on the Top-20 Findings on 5 Projects and Root Causes for False Positives.

Detector	Reviewed Findings	Confirmed Misuses	Precision	Kappa Score	Frequencies of Root Causes for False Positives						
					Uncommon	Analysis	Alternative	Inside	Dependent	Bug	Multiplicity
JADET	39	4	10.3%	0.97	21	3	8	0	1	0	2
GROUMINER	66	0	0.0%	0.97	25	22	8	7	2	1	1
DMMC	81	8	9.9%	0.91	9	19	18	19	4	4	0
TIKANGA	44	5	11.4%	0.93	18	7	7	0	7	0	0
Total	230	17		0.94	73	51	41	26	14	5	3

in Section 4 also used relative thresholds. Future work should investigate how these two alternatives compare.

O3: Particular usages may be uncommon without violating API constraints. Neglecting this causes 73 (34.3%) of the detectors' false positives in their top-20 findings. This calls for research on detecting patterns without setting a hard threshold on occurrence frequencies. Meanwhile, relaxing requirements on the co-occurrence of getter methods might reduce false positives significantly.

2. Analysis. The detectors use static analysis to determine the facts that belong to a particular usage. Imprecisions of these analyses lead to false positives. For example, the detectors mistakenly report five missing elements in code that uses multiple aliases for the same object and another 17 in code with nested control statements. In both cases, the analysis failed to capture all calls belonging to the same usage. GROUMINER reports two missing method calls, because it cannot resolve the receiver types in the chained calls and, therefore, fails to match a call between the pattern and the usage. Another example is that the detectors report eight missing method calls due to chained calls on a fluent API, such as `StringBuilder`, where their analyses cannot determine that all calls actually happen on the same object. JADET, GROUMINER, and DMMC together report nine missing calls that happen transitively in a helper method of the same class or through a wrapper object, such as a `BufferedStream`. DMMC reports a missing call that is located in the enclosing method of an anonymous class instance and a missing `close()` call on a parameter that is, by contract, closed by the callers. Moreover, GROUMINER reports four missing conditions that are checked by assertion helper methods. An inter-procedural detection strategy, as proposed by PR-MINER [11], could mitigate this problem.

O4: Imprecisions of the detectors' static analyses cause 51 (23.9%) of the false positives in their top-20 findings. An inter-procedural detection strategy might be able to eliminate 14 (6.6%) of these false positives.

3. Alternative. The detectors often learn a pattern and then report instances of alternative usages as violations. We define *alternative usages* as a different functionally correct way to use an API, either to achieve the same or a different functionality. Note that multiple alternatives may occur frequently enough to induce patterns. For example, JADET, TIKANGA, and DMMC learn that before a call to `next()`, there should always be a call to `hasNext()` on an `Iterator`. Consequently, they report 16 violations in usages that check either `isEmpty()` or `size()` on the underlying collection before fetching only the first element through the `Iterator`. DMMC reports another violation, because `isEmpty()` is used instead of `size()` before accessing a `List`. Another example is that JADET, TIKANGA, and DMMC learn that collections are filled one element at a time, e.g., by calling `add()`, and report 10 missing methods in usages that populate a collection differently, e.g., through the constructor or using `addAll()`. GROUMINER reports four usages where an alternative control statement is used, e.g., a `for` instead of a `while`.

A special case of this root cause is alternatives to obtain an instance of a type. For example, GROUMINER mistakenly reports two missing constructor calls where the instance is not created through a constructor call as in the pattern, but returned from a method call. JADET and DMMC each report one missing constructor call where an instance is not created, but passed as a parameter. While handling alternative patterns is an open problem, some tools such as ALATTIN already propose possible solutions [18].

O5: A violation of a pattern might be an instance of an alternative, correct way to use the respective API. Not considering this causes 41 (19.2%) of the false positives in their top-20 findings.

4. Inside. Objects that are stored in fields are often used across multiple methods of the field's declaring class. The respective API usages inside the individual methods might then deviate from usage patterns without being actual misuses. Figure 1 shows an example of such a case, where two fields of type `Iterator`, `in` and `out`, are used to implement the class `NeighborIterator`. When `in` yields no more elements (Line 12), the call to `next()` in Line 14 happens on `out` without a prior check whether it has more elements. While this appears to be a misuse of the `Iterator` API inside the enclosing method, it is a correct usage inside the enclosing class, since `NeighborIterator` itself implements `Iterator` and, thereby, inherits its usage constraints. Correct usages of `NeighborIterator` need to check its `hasNext()` method (Line 6) before calling its `next()` method (Line 11), which ensures that `out` has more elements when `next()` is called on it. DMMC and GROUMINER report sixteen violations for such usages of fields of a class.

A special case of this root cause is when a class uses part of its own API in its implementation. For example, when a `Collection` calls its own `add()` method in the implementation of its `addAll()` method. DMMC and GROUMINER report four such violations. This is particularly interesting, because these are actually self usages of the API, while the detectors target client usages. Since any codebase likely contains such self usages, detectors should consider this.

O6: The implementation code of a class may contain partial usages of the class' own API or fields. Such usages cause 26 (12.2%) of the detectors' false positives in their top-20 findings.

```
class NeighborIterator implements Iterator<GraphNode> {
  private final Iterator<DiGraphEdge> in = ...;
  private final Iterator<DiGraphEdge> out = ...;

  @Override
  public boolean hasNext() {
    return in.hasNext() || out.hasNext();
  }

  @Override
  public GraphNode next() {
    boolean isOut = !in.hasNext();
    Iterator<DiGraphEdge> curIterator = isOut ? out : in;
    DiGraphEdge s = curIterator.next();
    return isOut ? s.getDestination() : s.getSource();
  }

  ...
}
```

Figure 1: Correct Usages of `Iterator` Instances in the CLOSURE Project that Violate Usage Patterns.

5. Dependent. When two objects' states depend upon each other, usages sometimes check the state of one and implicitly draw conclusions about the state of the other. The detectors do not consider such inter-dependencies. For example, when two collections are maintained in parallel, i.e., always have the same size, it is sufficient to check the size of one of them before accessing either. The detectors falsely report 14 missing size checks in such usages. In 10 of these cases, the equal size is ensured by construction of the collections in the same method. In the remaining four cases, it is ensured elsewhere in the same class. We consider this a dangerous practice, because should the dependency between the collections ever change, it is easy to miss some of the code that relies on it. Thus, warning developers might be justified. Nevertheless, we count these cases as false positives, since the current usages are correct.

O7: Semantic dependencies between objects' states may implicitly ensure conditions. Not considering such inter-dependencies causes 14 (6.6%) of the detectors' false positives in their top-20 findings.

6. Multiplicity. The detectors cannot handle methods that may be called arbitrarily often. GROUMINER and JADET both learn a pattern where the `append()` method of `StringBuilder` is called twice and falsely report three missing method calls where it is called only once.

O8: Detectors should distinguish methods that require a specific number of calls, from methods that require one or more calls, and methods that may be called arbitrarily often. Not considering this causes 3 (1.4%) of the detectors' false positives in their top-20 findings.

7. Bug. A few findings are likely caused by mistakes in the detector implementations. DMMC reports four violations with an empty set of missing methods. These empty sets are produced when none of the potentially missing methods match DMMC's prevalence criteria. DMMC should probably filter such empty-set findings before reporting. GROUMINER reports one missing `if` that actually appears in all respective usages, because its graph mapping does not match the respective `if` node from one of the usages with the corresponding nodes of all the other usages.

7.2 Experiment RUB

We run all detectors to see which of the 64 known misuses from MUBENCH they can detect when given the respective crafted correct usages for pattern mining. Table 8 shows the results per detector. The second and third columns show the number of potential hits and the number of actual hits, after resolving disagreements. The fourth and fifth columns show the detectors' empirical recall upper bound and conceptual recall upper bound, respectively. The sixth column shows the Kappa score for the manual reviews. The remaining columns show the frequencies of root causes for divergences between a detector's conceptual capabilities from Table 3 and its actual findings in this experiment.

We find that GROUMINER has by far the best recall upper bound and also shows the best recall in Experiment RUB. This suggests that its graph representation is a good choice to capture the differences between correct usages and patterns. However, the gap between GROUMINER's conceptual upper bound recall and its empirical recall upper bound is quite noticeable. Actually, Table 8 shows that all four detectors fall considerably short of their conceptual recall upper bound in practice.

Generally, we observe two kinds of divergences between the actual findings and the conceptual capabilities: *Unexpected false negatives*, i.e., misuses that a detector should be able to detect, but does not, and *unexpected hits*, i.e., misuses that a detector supposedly cannot detect, but does. We investigate the root causes of each divergence to identify actionable ways to improve detectors.

O9: All detectors' empirical recall upper bound is much lower than their conceptual recall upper bound. Detectors' findings frequently diverge from their conceptual capabilities.

The Kappa scores indicate good reviewer agreement, albeit a little lower than in Experiment P. Since we only reviewed potential hits, i.e., findings in the same method as a known misuse, many potential hits were related to the known misuses. Consequently, we had several disagreements on whether a particular potential hit actually identifies a particular misuse. In total, we had 18 such disagreements (JADET: 4; GROUMINER: 6; DMMC: 5; TIKANGA: 3), which led us to formulate the lenient review process described in Section 5.2. We decided in favor of the detectors in eight of these cases. We observe that the Kappa score is a little lower for JADET, compared to the other detectors. Since the absolute number of disagreements is comparable and JADET had relatively few potential hits, i.e., a small number of decisions as a basis for the Kappa score, we attribute the lower score to chance.

Unexpected False Negatives

1. Representation. Current usage representations are not expressive enough to capture all details that are necessary to differentiate between misuses and correct usages. For example, DMMC and GROUMINER encode methods by their name only and, therefore, cannot detect a missing method call, when the usage calls an overloaded version of the respective method. For example, assume that a pattern requires a call to `getBytes(String)`, but the target usage calls `getBytes()` instead. An ideal misuse detector would still report a violation, since the expected method, with the correct parameters, is not called. However, since only the method name is used for comparison in both these detectors, such a violation is not detected. Another example is that, to use a `Cipher` instance for decryption, it must be in decrypt mode. This state condition is ensured by passing the constant `Cipher.DECRYPT` to the `Cipher`'s `init()` method. None of the detectors captures this way of ensuring that the condition holds, because they do not encode method-call arguments in their representations.

Table 8: Experiment RUB: Recall of the Isolated Detection Strategies and Root Causes for Divergences.

Detector	Potential Hits	Actual Hits	Empirical Recall Upper Bound	Conceptual Recall Upper Bound	Kappa Score	Frequencies of Root Causes					
						Representation	Matching	Analysis	Bug	Lenient	Exception Handling
JADET	19	15	23.4%	29.7%	0.76	4	4	1	0	3	2
GROUMINER	46	31	48.4%	75.0%	0.84	9	4	6	0	8	0
DMMC	40	15	23.4%	28.1%	0.85	5	0	0	2	5	0
TIKANGA	23	13	20.3%	29.7%	0.84	4	7	2	0	5	2
Total					0.83	22	15	9	2	21	4

```
1  writer.write(value);
```

```
1  try {
2    writer.write(value);
3  } finally {
4    if (writer != null)
5      writer.close();
6  }
```

Figure 2: Not Closing `Writer` vs. Correctly Closing `Writer`.

> *O10:* Inability to capture details necessary to differentiate misuses from correct usages in the usage representation is responsible for 22 (45.8%) of the unexpected false negatives.

2. Matching. The detectors fail to relate a pattern and a usage. Typically, detectors relate patterns and usages by their common facts. If there are no or only few common facts, detectors report no violation. For example, JADET's facts are pairs of method calls. In a scenario where `JFrame`'s `setPreferredSize()` method is accidentally called after its `pack()` method, JADET represents the usage with a pair (`pack`, `setPreferredSize`) and the pattern with the reverse pair. Since it compares facts by equality, JADET finds no relation between the pattern and the usage. Without common facts between a usage and a pattern, the detector assumes that these are two completely unrelated pieces of code and does not report a violation. Another example is when the pattern's facts relate to a type, e.g., `List` in `List.size()`, while the usage's facts relate to a super- or sub-type such as `ArrayList.size()` or `Collection.size()`. The detectors cannot relate these facts, since they are unaware of the type hierarchy. Also, TIKANGA misses four misuses, because the target misses more than two formulae of the pattern (TIKANGA's maximum distance for matching). For example, Figure 2 shows a misuse that does not close a `Writer` and the corresponding correct usage. In TIKANGA's representation, the difference between the misuse and the correct usage consists of three formulae: (1) that `close()` follows `write()` in case of normal execution, (2) that `close()` follows `write()` if the latter throws an exception, and (3) that `close()` is preceded by a `null` check.

> *O11:* When matching patterns and misuses, detectors should consider the semantics of their representation, e.g., call order and the number of usage facts generated by adding specific usage constructs, as well as code semantics, e.g., subtype relations. Neglecting this is responsible for 15 (31.3%) of the unexpected false negatives.

3. Analysis. The detectors rely on static analysis to extract their usage representations. Imprecisions in these analyses may obscure relations between patterns and usages. For example, GROUMINER fails to detect one missing `null` check, because

```
1  ArrayList markers;
2  if (layer == Layer.FOREGROUND) {
3    markers = (ArrayList) this.fgMarkers.get(index);
4  }
5  else {
6    markers = (ArrayList) this.bgMarkers.get(index);
7  }
8  // if (markers != null) { // <-- missing in misuse
9  boolean removed = markers.remove(marker);
10 // }
```

Figure 3: Example of an Analysis Problem of GROUMINER.

it cannot determine the receiver type for chained calls, such as for `m()` in `o.getX().m()`, which is not generally possible from source code alone. Also, it fails to detect another four missing `null` checks, because it overlooks dataflow dependencies. Figure 3 shows such a case. In addition to the `null` check, GROUMINER also misses the dataflow from the `get()` calls to the `remove()` call in the misuse, which makes the pattern and usage differ by multiple facts. GROUMINER, however, only reports a violation if the difference is a single fact. TIKANGA misses a call that occurs in the correct usage in one case and fails to capture the call order between two calls from the correct usage in another case. We assume that the cause is a limitation of its analysis, but could not ultimately verify this, because the tool's developer is not available to confirm the implementation details.

> *O12:* Imprecision of the analysis, which obscures the relation between patterns and misuses, causes 9 (18.8%) of the unexpected false negatives.

4. Bug. DMMC skips the comparison of a usage and a pattern if the pattern contains fewer calls than the usage, presumably to improve performance. The pattern for `AuthState` from Apache's HTTPCLIENT, for instance, requires three calls, of which the misuse scenario misses one. However, if this misuse has an additional, optional call that is not in the pattern, DMMC skips the comparison since now both the pattern and the target each contain 3 calls. This causes two unexpected false negatives in our experiment.

Unexpected Hits

1. Lenient. In all but two cases, the reason for *unexpected hits* is the lenient review process we use for Experiment RUB (see Section 5.2). In most cases, the detectors report a missing call that indicates a missing condition check. The only other case is that GROUMINER detects a missing context condition, in a scenario where some SWING code is required to run on the Event-Dispatching Thread (EDT). The delegation to the EDT is implemented by wrapping the code in an anonymous instance of `Runnable`, as shown in Figure 4. GROUMINER considers the code

```
public static void main(String[] args) {
  SwingUtilities.invokeLater(new Runnable() {
    public void run() {
      JFrame f = new JFrame("Main Window");
      // add components...
      f.setVisible(true);
    }
  });
}
```

Figure 4: Instantiating Swing Components on the Event-Dispatching Thread.

```
writer.write(value);
writer.close();
```

```
try {
  writer.write(value);
} finally {
  writer.close();
}
```

Figure 5: Closing `Writer` Without and With Exception Handling.

in `run()` as part of code of the enclosing method. Consequently, it suggests the misuse by reporting a missing instantiation of `Runnable` before the instantiation of the `JFrame`.

> *O13:* Missing method calls may indicate missing condition checks. Detectors that report these missing calls, despite not reporting the exact condition, find violations outside of their conceptual capabilities.

2. Exception Handling. In the remaining two cases, JADET and TIKANGA correctly report missing exception handling. For example, Figure 5 (left) shows a misuse where `close()` is not called when `write()` throws an exception. A corresponding correct usage is shown on the right. TIKANGA and JADET both represent the correct usage with two facts $\{(\texttt{write},\texttt{close}),(\texttt{write:EXC},\texttt{close})\}$, effectively encoding that `close()` is called after `write()` in normal execution and in case of an exception. In the misuse, they find the second fact missing. This capability of the implementation is not mentioned in the respective publications.

7.3 Experiment R

In Experiment R, we run all detectors to assess their recall without explicitly providing them with correct usages. In addition to MUBENCH's 64 misuses, we add the 14 new misuses from Experiment P and exclude the 25 hand-crafted examples for which there is no project code to mine patterns from. This leaves us with 53 misuses for Experiment R (Row 5 of Table 1).

Table 9 shows the results and Figure 6 visualizes the recall. JADET finds only the three misuses it already identified in Experiment P. GROUMINER does not find any of the misuses. TIKANGA finds the five misuses it already identified in Experiment P, one of the misuses that DMMC identified in Experiment P, and one of the misuses that JADET identified in Experiment P. DMMC finds two misuses from MUBENCH (both missing method calls), the eight misuses it reported in Experiment P, and one misuse both JADET and TIKANGA reported in Experiment P.

DMMC shows by far the best recall in Experiment R. This suggests that its relatively simple detection strategy works well when focusing on missing method calls. However, the recall of all detectors in the realistic setting offered by Experiment R is low. Analyzing the root causes for their bad performance, we identify two general problems with the design of the detectors and their evaluation setup.

1. Ranking. Experiment R shows that the detectors identify additional misuses beyond their top-20 findings that we considered in Experiment P. Unfortunately, they rank those misuses very low. For example, the two MUBENCH misuses DMMC finds are ranked 309 and 613. This is far beyond the number of findings that we can reasonably expect a user to assess. The four detectors in our experiments all use different ranking strategies, but none of the detectors from our survey in Section 4 compared different strategies on the same detector.

Table 9: Experiment R: Recall of the Detectors on MUBENCH and the New Misuses from Experiment P.

Detector	Potential Hits	Actual Hits	Recall	Kappa Score
JADET	4	3	5.7%	1.00
GROUMINER	4	0	0.0%	1.00
DMMC	25	11	20.8%	0.95
TIKANGA	9	7	13.2%	1.00
Total				0.97

> *O14:* Detectors need better ranking strategies to report true positives within their top findings. Furthermore, researchers should compare alternative ranking strategies for single detectors.

2. Usage Examples. The huge difference in the detectors' performance between Experiments RUB and R suggests that the cause is a shortage of correct usage examples in the target projects. One possibility is that the number of such examples is smaller than the detectors' minimal support for pattern mining, in which case we could simply lower these thresholds. However, this would likely also increase the number of false positives as the mined patterns generally become less reliable, which underlines the need to effectively filter false positives (*O1*) and improve ranking (*O14*). Another possibility is that no, or only very few, such examples exist in the projects. This would be a general problem with the evaluation setup of misuse detectors. To solve it, we need additional sources of usage examples to mine patterns from. Gruska *et al.* [37] demonstrated one possible approach by applying JADET in a cross-project setting with 6,000 projects, but did not measure recall. This strategy is also common in other recommender systems for software engineering, such as code-completion engines [50]. The misuse detectors CAR-MINER [17] and ALATTIN [18] implement an alternative approach, by specifically searching for usage examples of the APIs used in the target project via a code-search engine. Related to this, other lines of research proposed code-search engines to find usage examples in open source projects [51], [52] or on StackOverflow [53].

> *O15:* All detectors have low recall, likely due to lack of correct usage examples in target projects. Adoption of existing code-search techniques and cross-project mining could mitigate this problem.

The Kappa scores indicate mostly perfect reviewer agreement in Experiment R. This is because the detectors found almost exclusively the misuses that one of them also identified in Experiment P, i.e., the misuses we already agreed on before. The exception is DMMC, where we initially disagreed on one of its 14 potential hits for misuses from the original MUBENCH dataset.

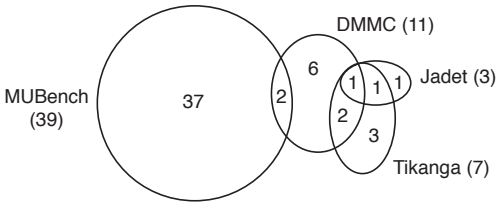

Figure 6: Recall of the Detectors in Experiment R

7.4 User Experience

We now report on our experiences as users of our subject misuse detectors. Our observations is based on the experience we gained while reviewing the detectors' findings in our experiments.

DMMC simply reports present and missing method calls, along with the source line number of the first present call. We find this output generally easy to interpret. The line number helps, especially, to locate usages in large methods. GROUMINER reports pattern and usage graphs, which are more difficult to understand. However, we find that the structural properties of the source code that the graph representation captures help with the interpretation. JADET and TIKANGA report the present and missing facts of their respective representations. We find that it is often difficult to relate the facts to each other, especially in the presence of multiple usages of the same API. This might be, in part, due to the textual representation we look at. While none of the detector implementations was intended to present their findings to end users, we still find it interesting to note that the challenge of explaining findings seems to correlate with the distance between the source code and the usage representation.

We also find that Bytecode-based detectors may report findings in code that the compiler introduces. For example, the compiler translates foreach loops into `Iterator` usages. TIKANGA reports a missing call in such a usage, i.e., it reports a missing call on `Iterator` in a method where `Iterator` does not appear in the source code. This finding confused us at first. While additional steps could be taken to assist the user in mapping such findings back to the source code, source-based detectors do not face this problem.

Our lenient review process shows that missing method calls frequently indicate missing conditions (*O13* and *O2*). While such findings do not report the entire problem, we found it relatively easy to deduce their meaning. In contrast, GROUMINER reports only a missing `if` node, when it captures a missing condition. While these findings more explicitly indicate the problem of a missing check, we feel that they are actually harder to act upon, because they give no information about *what* should be checked. This indicates a gap between a detector's capability to find a violation type and its ability to explain respective violations to users.

Above all, we believe that the detectors' precision is likely to be the biggest threat to their applicability in practice. As a previous study by Johnson *et al.* [41] shows, large numbers of false positives are a major barrier in the adoption of code analysis tools. This problem is made worse by the low recall of the detectors. Even if developers would take the time to review all reported warnings, they would still likely miss the vast majority of misuses.

7.5 Call to Action

We find that misuse detectors are practically capable of detecting a considerable part of the misuses in MUBENCH, when provided with the correct usages to compare to (Experiment RUB). However, even though the detectors are also capable of finding some misuses in a realistic setting (Experiments P and R), they suffer from extremely low precision (*O1*) and recall (*O15*). We identify four root causes for false negatives, seven root causes for false positives, and two general problems with the design of detectors and how they are typically evaluated. This leads us to several observations on how to advance the state-of-the-art in API-misuse detection. Therefore, we call researchers to action:

- We first need a precise definition of API usages, considering usage properties, such as the usage location (*O6*) and call multiplicities (*O8*).
- We need a representation of such usages that captures all code details necessary to distinguish correct usages from misuses (*O10*) and more precise analyses to identify usages in code (*O12* and *O4*).
- We need detectors that retrieve sufficiently many usage examples using project-external sources, such as large project sets or code-search engines (*O15*).
- We need detectors that go beyond the naive assumption that a deviation from the most-frequent usage corresponds to a misuse (*O3*), but consider program semantics, such as type hierarchies (*O11*) and implicit dependencies between objects (*O7*). We hypothesize that probabilistic models might be a way to tackle this problem.
- We need strategies to properly match patterns and usages in the presence of violations (*O9* and *O11*).
- We need strategies to properly handle alternative patterns for the same API (*O5*).
- Finally, we need good ranking strategies, to reduce the cost of reviewing findings (*O14*).

In order to achieve all this, we need repeatable and replicable studies that enable systematic evaluation and analysis of alternative approaches and strategies. We publish MUBENCH and MUBENCH-PIPE [24], as a foundation for such work, and call researchers to use and contribute to this infrastructure, to advance the state of the art in API-misuse detection.

8 THREATS TO VALIDITY

Construct Validity. Any detector's performance is dependent on its configuration. Due to the high effort of reviewing findings, we could not try different configurations for each detector. However, to give each detector a fair chance, we used the optimal configurations reported in the respective publications.

Our study focuses on static misuse detectors. Approaches based on dynamic analyses may perform differently and have unique strengths and weaknesses. To enable dynamic analyses of the project versions in MUBENCH, we would have to ensure that the respective code is executable (which requires a sufficient run-time environment, in addition to compile-time dependencies) and to provide example inputs for the execution. It is unclear how to do this such that it results in a fair comparison of both static and dynamic techniques, without resorting to comparing apples to oranges. In this work, we focused only on static approaches.

Our experiments focus on detectors that detect misuses in Java code. Therefore, the results may not generalize to detectors for other languages. We decided to focus on this subset of detectors, because the majority of approaches we identified in our survey targets Java. To include detectors that target other languages, we would have to either migrate them to Java or build up additional datasets for the respective languages, both of which is outside the scope of this work.

Internal Validity. Reviewing the detectors' findings was done by three of the authors and was not blind (i.e., we knew the detectors we were reviewing findings for). We could not do blind reviewing, because each approach has a distinct representation of usages and violations that cannot be anonymized. Moreover, two of the authors of this work are among the original authors of GROUMINER. We did our best to review objectively. To avoid bias, every finding was independently reviewed by two authors and for all findings of GROUMINER, at least one review was done by an author who was not involved in the original work.

While we did ask the original authors to confirm our assessment of the conceptual capabilities of their tools, we did not ask them to confirm the empirical results of our experiments. We estimate that, including discussions to resolve disagreements, it required each reviewer on average 2 minutes to verify whether a detector identified one of the known misuses in Experiments RUB and R and 5 minutes to verify whether a detector's finding identifies an actual misuse in Experiment P, where we needed to understand the respective code, check documentation, and sometimes also look into transitively called methods. This amounts to 24.8 hours of review effort per reviewer, 4 hours for JADET, 7.2 hours for GROUMINER, 4.7 hours for TIKANGA, and 8.9 hours for DMMC. We decided it is unreasonable to expect the original authors to invest this amount of time in verifying our assessments. We do, however, publish all our review data [10] to allow them and others to revisit our decisions.

External Validity. There may be violation categories we miss in MUC. The MUBENCH dataset may also not have enough examples of all violations. This may impact the detectors' comparisons. However, the existing MUBENCH dataset is based on over 1,200 reports from state-of-the-art bug datasets as well as developer input [3] and the results of two empirical studies on API usage directives. Our survey of existing detectors' capabilities also includes 12 detectors. This makes it unlikely that we miss a prevalent violation category.

Our dataset may not be representative of all possible real-world API misuses, especially, because we could only compile 29 (52%) of the 55 project versions and had to exclude the misuses in the other versions from our experiments. Compiling arbitrary versions of projects from the source control history of project is a challenging task. We invested two full weeks work of one of the authors and additional 3 months work of a student, to include as many project versions as possible. Still, loosing the examples for which we could not compile the respective project versions may bias the results of our experiments.

Ideally, our experiments would include thousands of misuses from a large number of projects and in each individual project version, to give us greater confidence in the generalizability of our results. However, currently, there is no such dataset. We invested several months of effort to collect and prepare MUBENCH in its current state, to make a first step towards a large benchmark. Now that the we have the infrastructure in place, it is straightforward to extend MUBENCH with misuse examples from different sources.

We publish MUBENCHPIPE and MUBENCH [24] and encourage others to extend the dataset and repeat our experiments, also with other detectors and detector configurations.

9 CONCLUSIONS

API-misuse detectors help developers write better software by warning them about potential misuses in their code. Despite the existence of many such detectors, there has been no attempt to systematically study types of API misuses and design detectors accordingly. In this paper, we addressed this gap by creating MUC, based on a dataset of 100 misuses. By evaluating the conceptual capabilities of 12 existing detectors against MUC, we identified shortcomings qualitatively. We then developed an automated benchmark pipeline, MUBENCHPIPE, to empirically evaluate four existing detectors. Our results reveal that misuse detectors are practically capable of detecting misuses, when explicitly provided with correct usages to mine patterns from. However, they suffer from extremely low precision and recall in a realistic application setting. We identify four root causes for false negatives, seven root causes for false positives, and two general problems with the design of detectors and the commonly-used evaluation setup. These lead us to several observations on how to advance the state-of-the-art in API-misuse detection in future work. We publish all our tooling and our dataset [24] to encourage other researchers to join us along this path.

ACKNOWLEDGEMENTS

We thank our students M. Kämmerer and J. Schlitzer for their work on MUBENCHPIPE and their help preparing MUBENCH, M. Monperrus for providing DMMC, A. Zeller and A. Wasylkowski for providing JADET and TIKANGA, and M. Pradel for additional examples for MUBENCH.

This work was partially funded by the German Federal Ministry of Education and Research (BMBF) within the Software Campus project *Eko*, grant no. 01IS12054, by the DFG as part of CRC 1119 CROSSING, and by the Hessen State Ministry for Higher Education, Research and the Arts (HMWK) within CRISP. The authors assume responsibility for the paper content.

REFERENCES

[1] M. Monperrus and M. Mezini, "Detecting missing method calls as violations of the majority rule," *ACM Transactions on Software Engineering and Methodology*, vol. 22, no. 1, pp. 1–25, 2013.

[2] J. Sushine, J. D. Herbsleb, and J. Aldrich, "Searching the state space: A qualitative study of API protocol usability," in *Proceedings of the 23rd IEEE International Conference on Program Comprehension*, ser. ICPC '15. IEEE Computer Society Press, 2015, pp. 82–93.

[3] S. Amann, S. Nadi, H. A. Nguyen, T. N. Nguyen, and M. Mezini, "MUBench: A benchmark for API-misuse detectors," in *Proceedings of the 13th Working Conference on Mining Software Repositories*, ser. MSR '16. ACM Press, 2016.

[4] S. Fahl, M. Harbach, T. Muders, L. Baumgärtner, B. Freisleben, and M. Smith, "Why Eve and Mallory love Android: An analysis of Android SSL (in)security," in *Proceedings of the 19th ACM Conference on Computer and Communications Security*, ser. CCS '12. ACM Press, 2012, pp. 50–61.

[5] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel, "An empirical study of cryptographic misuse in Android applications," in *Proceedings of the Conference on Computer & Communications Security*, ser. CCS'13. ACM Press, 2013, pp. 73–84.

[6] S. Nadi, S. Krüger, M. Mezini, and E. Bodden, ""Jumping through hoops": Why do developers struggle with cryptography APIs?" in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE'16. ACM Press, 2016.

[7] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov, "The most dangerous code in the world: Validating SSL certificates in non-browser software," in *Proceedings of the 19th ACM Conference on Computer and Communications Security*, ser. CCS '12. ACM Press, 2012, pp. 38–49.

[8] U. Dekel and J. D. Herbsleb, "Improving API documentation usability with knowledge pushing," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09. IEEE Computer Society Press, 2009, pp. 320–330.

[9] Y. Acar, M. Backes, S. Fahl, D. Kim, M. L. Mazurek, and C. Stransky, "You get where you're looking for. The impact of information sources on code security," in *Proceedings of the 37th IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 2016.

[10] "Artifact Page," 2017. [Online]. Available: http://www.st.informatik.tu-darmstadt.de/artifacts/mustudy/

[11] Z. Li and Y. Zhou, "PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code," in *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. ESEC/FSE '13. ACM Press, 2005, pp. 306–315.

[12] C. Lindig, "Mining patterns and violations using concept analysis," Universität des Saarlandes, Saarbrücken, Germany, Tech. Rep., 2007.

[13] A. Wasylkowski, A. Zeller, and C. Lindig, "Detecting object usage anomalies," in *Proceedings of the 6th ACM Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC/FSE '07. ACM Press, 2007, pp. 35–44.

[14] M. K. Ramanathan, A. Grama, and S. Jagannathan, "Static specification inference using predicate mining," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '07. ACM Press, 2007, pp. 123–134.

[15] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen, "Graph-based mining of multiple object usage patterns," in *Proceedings of the 7th ACM Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC/FSE '09. ACM Press, 2009, pp. 383–392.

[16] M. Acharya and T. Xie, "Mining API error-handling specifications from source code," in *Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, ser. FASE '09. Springer-Verlag GmbH, 2009, pp. 370–384.

[17] S. Thummalapenta and T. Xie, "Mining exception-handling rules as sequence association rules," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09. IEEE Computer Society Press, 2009, pp. 496–506.

[18] ——, "Alattin: Mining alternative patterns for detecting neglected conditions," in *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '09. IEEE Computer Society Press, 2009, pp. 283–294.

[19] A. Wasylkowski and A. Zeller, "Mining temporal specifications from object usage," *Automated Software Engineering*, vol. 18, no. 3-4, pp. 263–292, 2011.

[20] T. T. Nguyen, H. V. Pham, P. M. Vu, and T. T. Nguyen, "Recommending API usages for mobile apps with Hidden Markov Model," in *Proceedings of the 30th ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '15. IEEE Computer Society Press, 2015, pp. 795–800.

[21] O. Legunsen, W. U. Hassan, X. Xu, G. Roşu, and D. Marinov, "How good are the specs? A study of the bug-finding effectiveness of existing Java API specifications," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '16. ACM Press, 2016, pp. 602–613.

[22] M. Monperrus, M. Eichberg, E. Tekes, and M. Mezini, "What should developers be aware of? An empirical study on the directives of API documentation," *Empirical Software Engineering*, vol. 17, no. 6, pp. 703–737, 2012.

[23] C. DiBona, "Bidding farewell to Google Code." [Online]. Available: http://google-opensource.blogspot.com/2015/03/farewell-to-google-code.html

[24] "MUBench," 2017. [Online]. Available: https://github.com/stg-tud/MUBench/

[25] M. Pradel and T. R. Gross, "Leveraging test generation and specification mining for automated bug detection without false positives," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12. IEEE Computer Society Press, 2012, pp. 288–298.

[26] Q. Luo, Y. Zhang, C. Lee, D. Jin, P. O. Meredith, T. F. Şerbănuţă, and G. Rosu, "RV-Monitor: Efficient parametric runtime verification with simultaneous properties," in *Runtime Verification*. Springer-Verlag GmbH, 2014, pp. 285–300.

[27] M. Pradel, C. Jaspan, J. Aldrich, and T. R. Gross, "Statically checking API protocol conformance with mined multi-object specifications," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12. IEEE Computer Society Press, 2012, pp. 925–935.

[28] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf, "Bugs as deviant behavior: A general approach to inferring errors in systems code," in *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, ser. SOSP '01. ACM Press, 2001, pp. 57–72.

[29] "IEEE standard classification for software anomalies," *IEEE Std 1044-2009 (Revision of IEEE Std 1044-1993)*, pp. 1–23, 2010.

[30] R. Chillarege, I. S. Bhandari, J. K. Chaar, M. J. Halliday, D. S. Moebus, B. K. Ray, and M.-Y. Wong, "Orthogonal defect classification-a concept for in-process measurements," *IEEE Transactions on Software Engineering*, vol. 18, no. 11, pp. 943–956, 1992.

[31] M. Beller, R. Bholanath, S. McIntosh, and A. Zaidman, "Analyzing the state of static analysis: A large-scale evaluation in open source software," in *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering*, vol. 1. IEEE Computer Society Press, 2016.

[32] B. G. Glaser and A. L. Strauss, *The Discovery of Grounded Theory: Strategies for Qualitative Research*. Aldine de Gruyter, 1967, vol. 46.

[33] M. P. Robillard, E. Bodden, D. Kawrykow, M. Mezini, and T. Ratchford, "Automated API property inference techniques," *IEEE Transactions on Software Engineering*, vol. 39, no. 5, pp. 613–637, 2013.

[34] M. K. Ramanathan, A. Grama, and S. Jagannathan, "Path-sensitive inference of function precedence protocols," in *Proceedings of the 29th International Conference on Software Engineering*, ser. ICSE '07. IEEE Computer Society Press, 2007, pp. 240–250.

[35] B. Ganter and R. Wille, *Formal Concept Analysis: Mathematical Foundations*, 1st ed. Springer-Verlag New York, Inc., 1997.

[36] M. Monperrus, M. Bruch, and M. Mezini, "Detecting missing method calls in object-oriented software," in *Proceedings of the 24th European Conference on Object-oriented Programming*, ser. ECOOP '10. Springer-Verlag GmbH, 2010, pp. 2–25.

[37] N. Gruska, A. Wasylkowski, and A. Zeller, "Learning from 6,000 projects," in *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ser. ISTA '10. ACM Press, 2010, pp. 119–129.

[38] S. Brin, R. Motwani, J. D. Ullman, and S. Tsur, "Dynamic itemset counting and implication rules for market basket data," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '97. ACM Press, 1997, pp. 255–264.

[39] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, R. Stata, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata, "Extended static checking for Java," *ACM SIGPLAN Notices*, vol. 37, no. 5, pp. 234–245, 2002.

[40] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler, "A few billion lines of code later: Using static analysis to find bugs in the real world," *Communications of the ACM*, vol. 53, no. 2, pp. 66–75, 2010.

[41] B. Johnson and Y. Song, "Why don't software developers use static analysis tools to find bugs?" in *Proceedings of the 35th International Conference on Software Engineering*, ser. ICSE '13. IEEE Computer Society Press, 2013.

[42] G. Inc., "Closure compiler," 2017. [Online]. Available: https://developers.google.com/closure/compiler/

[43] iText Software, "iText, a Java PDF Library," 2017. [Online]. Available: https://sourceforge.net/projects/itext/

[44] R. U. i. N. Digital Security group, "JMRTD: An Open Source Java Implementation of Machine Readable Travel Documents," 2017. [Online]. Available: http://jmrtd.org/

[45] T. J. project, "Joda-Time," 2017. [Online]. Available: http://www.joda.org/joda-time/

[46] T. A. S. Foundation, "Apache Lucene," 2017. [Online]. Available: https://lucene.apache.org/core/

[47] S. Lu, Z. Li, F. Qin, L. Tan, P. Zhou, and Y. Zhou, "BugBench: Benchmarks for evaluating bug detection tools," in *In Workshop on the Evaluation of Software Defect Detection Tools*, 2005.

[48] C. Cifuentes, C. Hoermann, N. Keynes, L. Li, S. Long, E. Mealy, M. Mounteney, and B. Scholz, "BegBunch: Benchmarking for C bug detection tools," in *Proceedings of the International Workshop on Defects in Large Software Systems*, ser. DEFECTS '09. ACM Press, 2009, pp. 16–20.

[49] M. A. Saied, O. Benomar, H. Abdeen, and H. Sahraoui, "Mining multi-level API usage patterns," in *Proceedings of the 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering*, ser. SANER '15. IEEE Computer Society Press, 2015, pp. 23–32.

[50] S. Proksch, J. Lerch, and M. Mezini, "Intelligent code completion with Bayesian networks," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 25, no. 1, pp. 1–31, 2015.

[51] M. Grechanik, C. Fu, Q. Xie, C. McMillan, D. Poshyvanyk, and C. Cumby, "A search engine for finding highly relevant applications," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE '10. ACM Press, 2010, pp. 475–484.

[52] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu, "Portfolio: Finding relevant functions and their usage," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. ACM Press, 2011, pp. 111–120.

[53] L. Ponzanelli, G. Bavota, M. Di Penta, R. Oliveto, and M. Lanza, "Mining StackOverflow to turn the IDE into a self-confident programming prompter," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR '14. ACM Press, 2014, pp. 102–111.

Sven Amann is a doctoral candidate at TU Darmstadt, Germany. His primary research domain is API-misuse detection using static analyses and machine-learning techniques applied to examples mined from large code repositores and code search engines. Sven is founder and project lead of the MUBench benchmark suite. More on http://sven-amann.de.

Hoan Nguyen is a post-doctoral researcher at Iowa State University. He received his PhD from Iowa State University. His research interests include program analysis, software evolution and maintenance, and mining software repositories.

Sarah Nadi is an Assistant Professor at the University of Alberta. She received both her MMath (2010) and PhD (2014) degrees from the University of Waterloo, Canada, and spent approximately two years as a post-doctoral researcher at TU Darmstadt, Germany. Her research interests include mining software repositories, software product lines, and helping developers use APIs correctly.

Tien Nguyen is an associate professor at The University of Texas at Dallas. His research interests include program analysis, large-scale mining software repositories, and statistical approaches in software engineering.

Mira Mezini received the diploma degree in computer science from the University of Tirana, Albania, and the PhD degree in computer science from the University of Siegen, Germany. She is a professor of computer science at the Technische Universität Darmstadt, Germany, where she heads the Software Technology Lab.