A Systematic Evaluation of 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 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 API-misuse detectors along the same dimensions, and with 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. Our work provides these and several other novel insights that enable more powerful API-misuse detectors.

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

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 call close() after opening a FileOutputStream and writing to it, in order to avoid data loss or resource leaks. 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 can help, it is often not enough, 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 Stack Overflow, over official API documentation, even though the former promote many insecure API usages [9]. We confirm this tendency for other non-security informal references, such as Stack Overflow, over official API documentation, even though the former promote many insecure API usages [9]. We refer to those tools as API-misuse detectors.

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. We refer to those tools as API-misuse detectors.

There have been many attempts to address the problem of API misuse. Existing API-misuse detectors commonly mine usage patterns, i.e., equivalent API usages that occur frequently, and then report any anomalies with respect to these patterns as potential misuses [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]. In order 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 detection tools 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 misuse detectors. In order to create such a taxonomy, we need a diverse sample of API misuses. In our previous 4-page data paper, we described MUBENCH, a data set 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 sample misuses needed to create a taxonomy. To cover the entire problem space of API misuses, we added further misuses to this data set by looking at examples and claimed capabilities from API-misuse publications, and 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’ authors.

The previous step provides us with a conceptual comparison of existing detectors. We also wanted to quantitatively and empirically compare these API-misuse detectors, by both their precision and recall. This is a challenging task, due to the different underlying mechanisms and representations used by each detector. 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 they rely on an unavailable platform, five target C/C++ code, and one targets Dalvik Bytecode, while our benchmark contains Java misuses. Experiment 1 measures the recall of the detectors with respect to the known misuses in MuBench and an “ideal” pattern-mining setting, where the correct usage is provided to the detectors for mining the pattern. This evaluates the detector’s detection capabilities in isolation. Experiment 2 evaluates the precision of the detectors “in-the-wild” where they mine patterns and detect violations in the projects in MuBench. Finally, Experiment 3 evaluates the recall of the detectors “in-the-wild,” against both the MuBench dataset and the detectors’ own confirmed findings from Experiment 2.

Our conceptual analysis shows many previously neglected aspects of API misuse, such as correct exception handling and superfluous calls. Our quantitative results show that misuse detectors are practically capable of detecting misuses, when provided with the correct pattern. 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 naïve 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, before all else, detectors need to learn better models of API usage, for example, by obtaining more usage examples from different sources, such as code-search engines, and considering program semantics, such as type hierarchies and implicit dependencies between API usages. These and several other 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, especially with respect to detectors’ recall, which is typically not evaluated, and the application of detectors to individual projects, which do 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 [23], and our artifact page [10] provides full details on our results.

2 The API-Misuse Classification (MuC)

In this section, we introduce the API-Misuse Classification (MuC), our taxonomy for API misuses and framework for the evaluation of the capabilities of API-misuse detectors.

2.1 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 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 of these constraints, we call it a misuse, otherwise a correct usage.

The identification of API misuses has often been achieved through detecting deviant code [1], [11], [13], [15], [19], [24]. 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 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], [24].

In our previous work [3], we collected a data set of Java API misuses by reviewing bug reports of 30 real-world projects and surveying developers about API misuses. The data set we collected contains 90 misuses, 73 misuses from the real-world projects and 17 from the survey. For each real-world misuse, the data set 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.

2.2 The Classification

We introduce the API-Misuse Classification (MuC), a classification of misuses with respect to the violation type and API-usage elements involved, which allows us to systematically compare the capabilities of existing misuse detectors. To develop MuC, we went through all the misuses in MuBench and extracted their characteristics. We then iteratively came up with categories for these characteristics until we had a taxonomy by which we could describe all the misuses in the data set. Specifically, we found two dimensions whose intersection can be used to describe all misuses: the involved API-usage element and the violation type. In this section, we explain these dimensions and introduce the terminology we use in MuC.

API-usage elements are the program elements that appear 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, synchronization conditions, context
conditions, and value or state conditions, because of their distinct properties.

The violation type describes how a given usage violates a given pattern. In MUBENCH we find three violation types: missing, superfluous, and misplaced.

We define a violation as a combination of a violation type and an API-usage element. For example, a violation could be a “missing method-call.” Finally, an API misuse (misuse, for short) is an API usage with one or more violations.

Note that there was a simple API-misuse classification in the short MSR’16 data paper on MUBENCH [3]. However, MUC is a more comprehensive and detailed classification. In addition to the original MUBENCH misuses, we consider the findings of two empirical studies on API usage directives [8], [22]. Many of the directives those studies identify can be thought of as usage constraints in our terminology and their violation, consequently, as misuses. The studies report directives corresponding to all constraints in our terminology and their violation, consequently, as misuses. The studies report directives corresponding to all violations that appear in MUBENCH and a few more, for which we did not find examples in the wild. To make MUBENCH more comprehensive, we add hand-crafted examples of misuses violating these other directives, which we derive from the examples in the studies. This gives us 10 additional misuses, resulting in a total of 100 misuses for our experiments. For simplicity, we subsequently refer to this extended data set as MUBENCH.

Table 1 shows MUC. The numbers in the cells denote how many misuses with a respective violation are in MUBENCH. Note that since a single misuse can have multiple violations, 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. Superfluous calls and missing exception handling are less frequent, but still prevalent, while we have only few or no examples for the other violations.

We now discuss the different violation categories shown in Table 1 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.

A well-known violation category is missing method calls, which come from usage constraints that mandate calls to certain methods. For example, when adding UI elements to a JFrame, one needs to call validate() for the change to become visible.

Another case are superfluous method calls, which come from constraints that restrict calls to certain methods. An example would be when remove() is called on a list that is currently being iterated over.

The third case are misplaced method calls, which come from constraints that mandate certain methods to be called in a specific order. For example, one needs to call validate() after (versus before) adding elements to a JFrame.

Conditions. Conditions are a means for client code to ensure constraints mandated by APIs and to vary usages depending on program inputs.

A violation category is missing conditions, which come from usage constraints that mandate the client code to ensure certain conditions. Note that there often are alternative ways to ensure some condition. For example, one can ensure that a collection is not empty by checking isEmpty(), by checking its size(), or by adding an element to it before use. Prominent examples of missing conditions are missing null checks, e.g., to ensure that a receiver or a parameter of a call is not null. Another case are missing value or state conditions, e.g., 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., when accessing a HashMap from multiple threads [22]. Finally, missing context conditions may also occur, e.g., GUI components in SWING should only be updated from the Event Dispatching Thread [8].

Another case are superfluous conditions, where a condition prevents part of a usage, e.g., a method call, along certain execution paths, allows part of a usage in cases where it should not be executed, or is simply redundant. We have superfluous null checks, e.g., when the check occurs only after a method has been invoked on the respective object. Another case are superfluous value or state conditions, e.g., when the code checks isEmpty on a collection that’s guaranteed to contain an element. In multi-thread environments, superfluous synchronization conditions may occur, e.g., when the code requests a lock that it already holds. Finally, superfluous context conditions may also occur, e.g., when a GUI-event handler of a SWING application dispatches execution to the Event Dispatching Thread (EDT), which is redundant, because handlers are always invoked on the EDT.

The third case, misplaced conditions, we find to be inapplicable. A condition could be misplaced in the sense that it is checked only after an instruction that requires it as a precondition, like, for example, when checking that a reference is not null only after invoking a method on it. However, we interpret this case as a missing condition for the respective method call and, possibly, a redundant, i.e., superfluous, condition afterwards. A condition could also be misplaced relative to another condition, in the sense that the prior check subsumes the later one. Again, we argue that this either results in the later condition being superfluous or in the prior one missing a precondition.

Iteration. Iteration is another means of interacting with APIs, used, in particular, with collections and 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.

A violation category is missing iterations, which come from constraints that mandate a condition to 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. Note that even if the code ensures the correct condition, say, with an if, this constraint is violated as long as the check and, depending on its outcome, the invocation of wait() are not repeated.

Another case is superfluous iterations, where part of a usage should execute at most once or when a reiteration 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.

The third case, misplaced iteration, we find to be inapplicable. Iteration always involves some instructions that are repeated, hence, either these instructions need to be repeated, in which case the iteration may be missing, or the instructions should not be repeated, in which case the iteration may be superfluous. Since iteration itself does not do anything—it is the instructions within the loop that compose the functionality—it can also not be misplaced relative to other iteration.

**Exception Handling.** Exceptions are a means for APIs to communicate errors to client code. It often depends on the specific API whether and how different errors should be handled.

A violation category is *missing exception handling*, which comes from errors that the client code should take actions to recover from. 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.

Another case is *superfluous exception handling*, which comes from intercepting 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.

The third case, misplaced exception handling, we find to be inapplicable. Exception handling happens always with respect to one or multiple instructions, hence, either these instructions require the handling, in which case it may be missing, or the instructions should not have such handling, in which case it may be superfluous. The only way in which exception handling could be misplaced is when an earlier catch clause is more general than a later one. For example, handling `IOException` before `FileNotFoundException` theoretically results in a missing file being handled like any other IO error. We argue that this is not a misuse of the API, but of the `try` language construct. Moreover, the Java compiler already rejects such unreachable catch clauses at compile time.

### 3 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. [25] and looked at all publications they referred 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 the MuC to guide the comparison. Table 2 summarizes the capabilities of each detector with respect to MuC, to provide a conceptual classification of the existing detectors. Note that while there are dynamic approaches for detecting API-usage problems (e.g., [26], [27]), we focus on static approaches. Our goal in this work is to conceptually and empirically evaluate misuse detectors.

It is very difficult to design a unified evaluation setup that fairly compares both static and dynamic techniques, without resorting to comparing apples to oranges. Therefore, we chose to focus only on static approaches. We use the published description and results of each detector to identify which of the 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.

**PR-Miner** is a misuse detector for C [17]. 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 interprocedural 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).

**COLIBRI/ML** is another misuse detector for C [12]. It reimplements PR-Miner using **Formal Concept Analysis** [28] 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.

**JADE** 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 relation. 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. The encoding of call-order relations allows JADE to detect misplaced calls in addition to missing 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 JADE to five target projects, thereby finding violations of project-specific patterns. The authors reviewed the top-10 violations reported per project and found 0-13% true positives. Other findings were classified as code smells or hints.

In a subsequent study, JADE was applied in a cross-project setting with 6,000 projects [29], using a minimum pattern support of 200. The experiment demonstrates the scalability of the approach, while achieving a true positive rate of 22% on the top 25% of 50 reported violations.

**RGJ07** is a misuse detector for C [14]. It encodes usages as...
Table 2: Capabilities of 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.

| Misuse Detector | Method Calls | Conditions | Ex. Handl. | Iteration |
|-----------------|--------------|------------|------------|-----------|
| PR-Miner [1]    | □            | □          |            |           |
| COLIBRI-Miner [2] | □          | □          |            |           |
| Jadex [12]      | □            | □          |            |           |
| RGJ07 [14]      | □            | □          |            |           |
| Chronicler      | □            | □          |            |           |
| GROUMiner [6]   | □            | □          |            |           |
| AX09 [16]       | □            | □          |            |           |
| CAR-Miner [17]  | □            | □          |            |           |
| Tikanga [20]    | □            | □          |            |           |
| DroidAnalyze [19] | □      | □          |            |           |

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/state conditions. Since patterns contain preceding calls on arguments, it may also detect misplaced calls and 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 discuss several examples of actual bugs their approach detects, but report no statistic on the detection performance.

**Chronicler** is a misuse detector for C [30]. 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 and misplaced method calls. Since loops are unrolled exactly once, it cannot detect missing iteration. The evaluation applied Chronicler to five projects, thereby finding violations of project-specific patterns. The authors compare the identified protocols with the documented protocols for one API and discuss a few examples of actual bugs found by their tool.

**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 and misplaced 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 and confirmed 12% as bugs. Others were classified as code smells.

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 and misplaced 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/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 findings and confirmed 80 – 93% as bugs.

**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 and misplaced 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 and confirmed 41-82% identify wrong error handling. Others were classified as hints to improve code quality.

**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/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 and confirmed that 12% and 52% identify API.

This allows it to detect missing, misplaced, and superfluous 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 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 \( s = 1 - |E(x)|/(|E(x)| + |A(x)|) \), 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 19 findings and confirmed 84% as bugs. Others were classified as code smells.

**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, misplaced, and superfluous method calls and even to suggest solutions for them. An evaluation of this mechanism is not provided in the respective paper.

**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 [31] 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 [28] to obtain patterns and violations at the same time. **Tikanga**'s capabilities are the same as **JADET**'s. The evaluation applied the approach to six projects individually, finding violations of project-specific patterns. The authors manually reviewed the top-25% of violations per project and confirmed up to 6% as bugs. Others were classified as code smells.

**DMMC** is a misuse detector for Java [1], 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 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 \( s = 1 - |E(x)|/(|E(x)| + |A(x)|) \), 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 19 findings and confirmed 84% as bugs. Others were classified as workarounds for bugs inside a used API.

**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, misplaced, and superfluous method calls and even to suggest solutions for them. An evaluation of this mechanism is not provided in the respective paper.

**Summary.** 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**, 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 superfluous elements, as a usage with such an element is never part of any pattern. The exception is **DROIDASSIST**, which might find superfluous calls as being unlikely.

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 and most may also identify misplaced 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 only exception is, again, **DROIDASSIST**, which uses a probabilistic approach. Since many approaches produce a high number of false positives, they propose 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.

Most evaluations apply detectors to target projects individually, where the number of projects ranges from one to nine, with a median of five projects. Most authors verified the top-X findings of their detectors, where X is a fixed number or percentage. The evaluations either present anecdotal evidence of true positives or measure the precision of detectors. It appears that the detectors that focus on specific violations, such as error handling or missing method calls, have higher precision. However, the variance in the evaluation settings makes a direct comparison of the detectors' performance unreliable. Moreover, the recall of API-misuse detectors has never been assessed.

### 4 Experimental Setup

Section 3 conceptually compared detectors’ capabilities. In this section, we describe the setup we use to experimentally compare their capabilities. To enable us to systematically compare various detectors on the same dataset, we build **MUBENCHPIPE**, a benchmark pipeline for API-misuse detectors on top of **MUBENCH**. This standardizes and automates much of the evaluation process to facilitate the reproduction of our study. It also enables us to compare multiple detectors on the same subjects, and enables adding new detectors to the comparison in the future. We now explain how we setup our evaluation using the new pipeline. We publish both the pipeline and the dataset [23].

#### 4.1 Subject Detectors

We use our benchmark to compare the performance of detectors. 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 [32]. We exclude **DROIDASSIST**, because its implementation only supports Dalvik.
Bytecode\textsuperscript{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.

4.2 Misuse Dataset

We use MuBench, described in Section 2,\textsuperscript{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.

4.3 Experiments

Our subject detectors all mine patterns that they use for detection. Failure to find a particular misuse can be due to (1) the detection strategy being unable to identify the misuse or (2) the mining approach being unable to identify a pattern for the respective correct usage. To cover both possibilities, we design three experiments as described below. Note that our survey in Section 3 shows that a per-project setup, where patterns mined from one project are used to detect misuses in the same project, is the most common setup for misuse detectors. Thus, we follow this setup for all three experiments.

Experiment 1

To assess the detection capabilities in isolation, we provide the detectors with examples of correct usage for pattern mining. This mimics a perfect pattern-mining stage that always finds the patterns required to identify the known misuses. This experiment gives us an upper bound to the practical capabilities of a detector. We calculate two numbers for each detector. The first is its recall, i.e., the fraction of misuses a detector actually finds from all the 64 known misuses in MuBench. The second is its conceptual recall upper bound, i.e., the fraction of the 64 misuses that match its capabilities from Table 2. An ideal detector should have a recall rate equal to its recall upper bound. Otherwise, its practical capabilities do not match its conceptual capabilities. In such cases, we investigate the root causes for such mismatches.

We manually derive the correct usages from the fixing commit recorded in MuBench for every misuse. 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.

To be able to run detectors that do pattern mining and detection at the same time, we provide all detectors with 50 copies of the crafted correct usage, along with a single file that contains the known misuse. We configure the detectors to mine patterns with a minimum support of 50, thereby ensuring that they mine patterns only from the code in our crafted correct usages. We ensure that all detectors consider each copy as a distinct usage. We chose 50 as a threshold, since it is high enough to ensure that no detector mines patterns from the misuse file itself.

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. After the review, any disagreements between the reviewers are discussed until a consensus is reached about whether the detector finds the misuse. We report the Kappa score as a measure of the reviewers' agreement. If at least one potential hit identifies the misuse, we count it as a hit. 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 still give a valuable hint about the problem.

Experiment 2

To evaluate the detector as a whole, we assess the capabilities of both pattern mining and detection in Experiment 2. To do so, we look at all findings of the detectors, not just potential hits with respect to known misuses, and provide an estimate of the precision of each detector. This is the common evaluation scenario for misuse detectors, where the detector mines patterns and detects violations on a per-project basis, and these violations are then manually reviewed to confirm if they are indeed misuses or not.

Since manually reviewing all findings of all detectors on all project versions is infeasible, we sample five project versions and review the top-20 findings per detector on each version, as determined by the detectors' individual ranking strategies. To ensure a fair selection of projects, we first run all detectors on all project versions. For practical reasons, we limit the individual runs of each detector on a project version to two hours. The run statistics are summarized in Table 3.

Jadet and Tikanga fail on one project version and DMMC fails on four project versions, since the Bytecode toolkit does 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, i.e., either all detectors report a relatively large or a relatively small number of findings on any given version. We hypothesise that the causes for this phenomenon may impact the ability of detectors to identify misuses in the project versions. Therefore, we sample the two projects with the highest average number of findings across all detectors (Closure [33] v319, iText [34] v5091) and the two projects with the lowest average number of findings across all detectors (JMRTD [35] v51 and Joda-Time [36] v1231). In additional, we randomly select one more project version (Apache Lucene [37] 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 likely share a lot of code, hence, detectors likely perform very similar on them.

Again, two authors independently review each finding from this sample and mark it as a misuse or not. To determine this, they consider the logic and the documentation in the source

---

\textsuperscript{1} A bytecode format developed by Google, which is optimized for the characteristics of mobile operating systems (especially for the Android platform).
Table 3: Number of Findings per Detector on All Compilable Project Versions in MuBench. Experiment 2 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               |                  |
|                       |         | Average            |                  |
|                       |         |                    |                  |
| Apache Commons Lang   | 587     | 0                  | 28               |
| Apache Commons Math   | 998     | 17 error           | 17               |
| ADEmpiere             | 1312    | 0                  | 27               |
| Alibaba DuiD          | e10f28  | 17 timeout         | 5                |
| Closure               | 114     | 113 101            | 24               |
| Closure               | 319     | 176 126            | 45               |
| Apache HttpClient     | 302     | 0                  | 12               |
| Apache HttpClient     | 444     | 0                  | 15               |
| Apache HttpClient     | 452     | 0                  | 12               |
| iText                 | 5091    | 17 198             | 55               |
| Apache JACKRabbit     | 1601    | 12                 | 186              |
| Apache JACKRabbit     | 1678    | 0                  | 15               |
| Apache JACKRabbit     | 1694    | 13                 | 186              |
| Apache JACKRabbit     | 1750    | 10 timeout         | 8                |
| JFreeChart            | 103     | 167 timeout        | 88               |
| JFreeChart            | 164     | 168 timeout        | 90               |
| JFreeChart            | 881     | 194 timeout        | 93               |
| JFreeChart            | 1025    | 194 timeout        | 93               |
| JFreeChart            | 2183    | 190 timeout        | 100              |
| JFreeChart            | 2266    | 195 timeout        | 102              |
| JMRTD                 | 51      | 0                  | 11               |
| JMRTD                 | 67      | 0                  | 10               |
| Joda-time             | 1231    | 0                  | 0                |
| Apache Lucene         | 207     | 0                  | 140              |
| Apache Lucene         | 754     | 0                  | 54               |
| Apache Lucene         | 1251    | 2                  | 62               |
| Apache Lucene         | 1918    | 2                  | 88               |
| Mozilla Rhino         | 286251  | error              | 55               |

Experiment 3

While Experiment 1 measures recall in an idealized mining setting, we also want to estimate recall in a realistic setting. 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. MuBenchPipe enables us to do this through Experiment 3. As the ground truth for this experiment, we use all known misuses from MuBench plus the true-positives identified by any of the detectors in Experiment 2. This means that Experiment 3 not only evaluates recall against the misuses of MuBench, but also practically cross-validates the detector capabilities against each other.

We run all detectors (with both mining and detection) on all projects versions and review all potential hits in the same process as for Experiment 1. While this does not give us their absolute recall, it provides their recall with respect to a large number of known misuses.

4.4 MuBenchPipe

Following the idea of automated bug-detection benchmarks for C programs, such as BugBench [38] and BegBunch [39], we facilitate the task of benchmarking multiple detectors on our misuse dataset with an evaluation pipeline. Therefore, we build MuBenchPipe, a benchmarking pipeline that automates many of our evaluation steps including retrieving and compiling the project versions’ source code in MuBench, running detectors, collecting their findings, and performing the manual reviews of potential hits. MuBenchPipe provides a command-line interface to control these tasks. The subsequently describe the pipeline steps we implemented to facilitate our evaluation and to enable easy replication of our experiments.

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 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. The pipeline can be run in Experiment 1 or Experiment 2.
mode. Technically, running Experiment 2 is the same as running Experiment 3. The difference comes in the reviewing process where only findings that match the known misuses are reviewed in Experiment 3, while a sample of all findings is reviewed in Experiment 2.

Validation. To help with the manual review of findings, MIUBENCHPipe automatically publishes experiment results to a review website [10] that shows for every detector finding 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 1 and 3, MIUBENCHPipe 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 2, MIUBENCHPipe shows all findings of the detector on the review site, each with the source code it is located in.

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 Kappa scores.

4.5 Reproduction, Replication, and Extension

MIUBENCHPipe comes with a Docker container, which allows running reproducible experiments across platforms, without the need to ensure a proper environment setup. Its review website—based on PHP and MySQL, such that it can be hosted on any off-the-shelf webspace—facilitates independent reviews, even when researchers work from different locations, while ensuring review integrity using PHP Basic Auth. The website may also directly be used as an artifact to publish review results and experiment statistics. MIUBENCHPipe defines a simple data schema for misuse examples to facilitate extensions of MIUBENCH. It also provides a convenient Java interface as a Maven dependency to enable plugging in additional detectors for evaluation on the benchmark. For further technical details on how to add a new detector, we refer to the project website [23].

5 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].

5.1 Experiment 1

We run all detectors to see which of the 64 known misuses from MIUBENCH they can detect when given the respective crafted correct usages for pattern mining. Table 4 shows the results per detector. The second and third column show the number of potential hits and the number of actual hits, after resolving disagreements. The fourth and fifth column show the detector’s recall and its conceptual upper bound of recall, 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 2 and its actual findings in this experiment. Note that two detectors sometimes have the same root cause for their respective divergence on the same misuse.

We find that GROUMiner has by far the best recall upper bound and also shows the best recall in Experiment 1. 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 actual recall is quite noticeable. Actually, Table 4 shows that all four detectors fail 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 misuse detectors.

O1: All detectors’ recall is much lower than their conceptual upper bound of recall and their findings frequently diverge from their conceptual capabilities.

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.

O2: 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 (misplaced method call), 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, because a usage of PrintWriter misses to close the writer (call
Figure 1: Example of an Analysis Problem of GROUMINER.

to close()) in both the normal and the exceptional case (try-finally), if it has been successfully created (null check before close()).

O3: 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 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 1 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 calls that occur in the pattern in one case and associations between facts that frequently co-occur 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.

O4: 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 our lenient review process described in the setup of Experiment 1 in Section 4. 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 2. GROUMINER considers the code 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.

O5: 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 3 (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 { (write,close),(write:EXC,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.

5.2 Experiment 2

We now investigate each detector’s precision by reviewing the top-20 findings per detector on each of our five sample projects. Table 5

Table 4: Experiment 1 Recall of the Isolated Detection Strategies and Root Causes for Divergences.

| Detector    | Potential Hits | Actual Hits | Recall | Recall Upper Bound | Kappa Score | Frequencies of Root Causes |
|-------------|----------------|-------------|--------|--------------------|-------------|---------------------------|
| JADET       | 19             | 15          | 23.4%  | 29.7%              | 76.4%       | 4 4 1 0 3 2               |
| GROUMINER   | 46             | 31          | 48.4%  | 75.0%              | 84.4%       | 9 4 6 0 8 0               |
| TIKANGA     | 23             | 13          | 20.3%  | 29.7%              | 83.9%       | 4 7 2 0 5 2               |
| DMMC        | 40             | 15          | 23.4%  | 26.6%              | 84.6%       | 5 0 0 2 5 0               |

Total 82.8% 22 15 9 2 21 4
In the following, we discuss these root causes summarized across all detectors, in 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 frequent 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 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 loops for respective invocations outside a loop, which are perfectly fine according to the API. Approaches such as multi-level patterns [40] or AGLATTIN’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 in Section 3 also used relative thresholds. Future work should investigate how these two alternatives compare.

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, where they fail to capture all calls belonging to a 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 intra-procedural 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

---

**Table 5: Experiment 2** 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 |
|------------|-------------------|-------------------|-----------|-------------|
| JADET      | 39                | 4                 | 10.3%     | 97.2%       |
| GROUMINER  | 66                | 0                 | 0.0%      | 97.0%       |
| TIKANGA    | 44                | 5                 | 11.4%     | 92.6%       |
| DMMC       | 81                | 8                 | 9.9%      | 90.8%       |
| **Total**  | 230               | 17                | **94.0%** |             |

| Misuse Type   | Uncommon | Analysis | Alternative | Inside | Dependent | Bug | Multiplicity |
|---------------|----------|----------|-------------|--------|-----------|-----|--------------|
| **JADET**     | 21       | 3        | 8           | 0      | 1         | 0   | 2            |
| **GROUMINER** | 25       | 22       | 8           | 7      | 2         | 1   | 1            |
| **TIKANGA**   | 18       | 7        | 7           | 0      | 7         | 0   | 0            |
| **DMMC**      | 9        | 19       | 18          | 19     | 4         | 4   | 0            |

1. ```java
   try {
     writer.write(value);
   }
   finally {
     writer.close();
   }
```
method of an anonymous class instance and a missing \texttt{close()} call on a parameter that is, by contract, closed by the callers. Moreover, \textsc{GROUMiner} reports four missing conditions that are checked by assertion helper methods. An inter-procedural detection strategy, as proposed by \textsc{PR-Miner} \cite{11}, could mitigate this problem.

\textit{O9:} 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. \textbf{Alternative.} The detectors often learn a pattern and then report instances of alternative usages as violations. We define alternative usages as a different 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, \textsc{JADET}, \textsc{Tikanga}, and \textsc{DMMC} learn that before a call to \texttt{next()} there should always be a call to \texttt{hasNext()} on an \texttt{Iterator}. Consequently, they report 16 violations in usages that only pull the first element and check either \texttt{isEmpty()} or \texttt{size()} on the underlying collection to ensure this element exists. \textsc{DMMC} reports another violation, because \texttt{isEmpty()} is used instead of \texttt{size()} before accessing a \texttt{List}. Another example is that \textsc{JADET}, \textsc{Tikanga}, and \textsc{DMMC} learn that collections are filled one element at a time, e.g., by calling \texttt{add()}, and report 10 missing methods in usages that populate a collection differently, e.g., through the constructor or using \texttt{addAll()}. \textsc{GROUMiner} reports four usages where an alternative control statement is used, e.g., a \texttt{for} instead of a \texttt{while}.

A special case of this root cause is alternatives to obtain an instance of a type. For example, \textsc{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. \textsc{JADET} and \textsc{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 \textsc{Alattin} already propose possible solutions \cite{18}.

\textit{O10:} A violation of one pattern might be an instance of another, alternative pattern. Not considering this causes 41 (19.2\%) of the false positives in their top-20 findings.

4. \textbf{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 4 shows an example of such a case, where two fields of type \texttt{Iterator}, \texttt{in} and \texttt{out}, are used to implement the class \texttt{NeighborIterator}. When \texttt{in} yields no more elements (Line 12), the call to \texttt{next()} in Line 14 happens on \texttt{out} without a prior check whether it has more elements. While this appears to be a misuse of the \texttt{Iterator} API inside the enclosing method, it is a correct usage inside the enclosing class, since \texttt{NeighborIterator} itself implements \texttt{Iterator} and, thereby, inherits its usage constraints. Correct usages of \texttt{NeighborIterator} need to check its \texttt{hasNext()} method (Line 6) before calling its \texttt{next()} method (Line 14), which ensures that \texttt{out} has more elements when \texttt{next()} is called on it. \textsc{DMMC} and \textsc{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 \texttt{Collection} calls its own \texttt{add()} method in the implementation of its \texttt{addAll()} method. \textsc{DMMC} and \textsc{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.

\textit{O11:} 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.

5. \textbf{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.

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

\textit{O13:} 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. \textbf{Bug.} A few findings are likely caused by mistakes in the detector implementations. \textsc{DMMC} reports four violations with an empty set of missing methods. These empty sets are produced when none

\begin{figure}
\centering
\begin{verbatim}
public class NeighborIterator implements Iterator\langle GraphNode \rangle {
    private final Iterator\langle DiGraphEdge \rangle in = ...;
    private final Iterator\langle DiGraphEdge \rangle out = ...;

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

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

    ...}
\end{verbatim}
\caption{Correct Usages of \texttt{Iterator} Instances in the \texttt{CLOSURE} Project that Violate Usage Patterns.}
\end{figure}
Table 6: Experiment 3: Recall of the Detectors on MuBench and the New Misuses from Experiment 2

| Detector | Potential Hits | Actual Hits | Recall | Kappa Score |
|----------|----------------|-------------|--------|-------------|
| Jadet    | 4              | 3           | 5.7\%  | 100.0\%     |
| GROUMINER | 4          | 0           | 0.0\%  | 100.0\%     |
| Tikanga  | 9              | 7           | 13.2\% | 100.0\%     |
| DMMC     | 25             | 11          | 20.8\% | 94.8\%      |

Total 96.9\%

Figure 5: Recall of the Detectors in Experiment 3

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.

5.3 Experiment 3

In Experiment 3, we run all detectors to assess their recall when using their own pattern mining. To MuBench’s 64 misuses we add the 14 new misuses from Experiment 2 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 3.

Table 6 shows the results and Figure 5 visualizes the recall. Jadet finds only the three misuses it already identified in Experiment 2. GROUMINER does not find any of the misuses. Tikanga finds the five misuses it already identified in Experiment 2, one of the misuses that DMMC identified in Experiment 2 and one of the misuses that Jadet identified in Experiment 2. DMMC finds two misuses from MuBench (both missing method calls), the eight misuses it reported in Experiment 2 and one misuse both Jadet and Tikanga reported in Experiment 2.

DMMC shows by far the best recall in Experiment 3. This suggests that its relatively simple detection strategy works well when focusing on missing method calls. However, the recall of all detectors in the real setting offered by Experiment 3 is low. Analyzing the root causes for their bad performance, we identify two general problems with the design of the detectors and their evaluation setting.

1. Ranking. While Experiment 3 shows that the detectors identify more misuses beyond their top-20 findings, they, unfortunately, rank those 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 3 compared different strategies on the same detector.

2. Usage Examples. The huge difference in the detectors’ performance between Experiments 1 and 3 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 (O15) 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 setting of misuse detectors. To solve it, we need additional sources of usage examples to mine patterns from. Gruska et al. 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. The misuse detectors CAR-MINER and ALATTIN 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 or on StackOverflow.

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.

5.4 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 \((O^2)\) and \((C^2)\). While such findings do not report the entire problem, we found it relatively easy to deduce their meaning. Contrarily, 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. \([55]\) 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, which implies that even if developers would take the time to review all reported warnings, they would still likely miss the vast majority of misuses.

### 5.5 Call to Action

We find that misuse detectors are practically capable of detecting a considerable part of the misuses in \(\text{MUBENCH}\), when provided with the correct usages to compare to \(\text{Experiment 1}\). However, even though the detectors are also capable of finding some misuses in a realistic setting (Experiments 2 and 3), they suffer from extremely low precision \((C^6)\) and recall \((C^7)\). 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 need detectors that retrieve sufficiently many usage properties, such as the usage location \((C^7)\) and call multiplicities \((C^2)\).**
- **We need a representation of such usages that captures all code details necessary to distinguish correct usages from misuses \((C^2)\) and more precise analyses to identify usages in code \((C^7)\) and \((C^9)\).**
- **We need detectors that retrieve sufficiently many usage examples using project-external sources, such as large project sets or code-search engines \((C^7)\).**
- **We need detectors that go beyond the naive assumption that a deviation from the most-frequent usage corresponds to a misuse \((C^8)\), but consider program semantics, such as type hierarchies \((C^7)\) and implicit dependencies between objects \((C^7)\). 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 \((C^7)\) and \((C^9)\).**
- **We need strategies to properly handle alternative patterns for the same API \((C^7)\).**
- **Finally, we need good ranking strategies, to reduce the cost of reviewing findings \((C^7)\).**

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 \([23]\) as a foundation for such work, and call researcher to use and contribute to this infrastructure, to advance the state of the art in API-misuse detection.

### 6 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 \(\text{MUBENCH}\), we would have to ensure that the respective code is executable (which requires a sufficient runtime 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. Therefore, we chose to focus only on static approaches.

Our experiments focus on detectors that detect misuses in Java code. Consequently, 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 \(\text{Experiment 1}\) and 3 and 5 minutes to verify whether a detector’s finding identifies an actual misuse in \(\text{Experiment 2}\), 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 \(\text{MUC}\). The \(\text{MUBENCH}\) dataset may also not have enough examples of all violations. This may impact the detectors’ comparisons. However, the existing \(\text{MUBENCH}\) 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 for API misuses in the wild, 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 2 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 data set. We invested several months of effort to collect and prepare MUBENCH in its current state, to make a first step towards it. Now that we have the infrastructure in place, it is straightforward to extend MUBENCH with misuse examples from different sources.

We publish MUBENCHPIPE and MUBENCH [23] and encourage others to extend the data set and repeat our experiments, also with other detectors and detector configurations.

7 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 setting. 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 [23] 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 JADE 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 Éko, 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, p. 1–25, June 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. Piscataway, NJ, USA: IEEE Press, 2015, pp. 82–93. [Online]. Available: http://dx.doi.org/10.1109/ICPC.2015.35

[3] S. Aman, 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, 2016. [Online]. Available: http://doi.acm.org/10.1145/2901739.2903506

[4] S. Fahí, M. Habruch, T. Maders, L. Baumgartner, 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. New York, NY, USA: ACM, 2012, pp. 50–61. [Online]. Available: http://doi.acm.org/10.1145/2382196.2382205

[5] M. Egele, D. Brunley, Y. Fratantonio, and C. Kruegel, “An empirical study of cryptograhic misuse in Android applications,” in Proceedings of the 2013 ACM SIGSAC Conference on Computer &#38; Communications Security, ser. CCS ’13. New York, NY, USA: ACM, 2013, pp. 73–84. [Online]. Available: http://doi.acm.org/10.1145/2508859.2516693

[6] S. Nadi, S. Krüger, M. Mezini, and E. Bodden, “Jumping through hoops”: Why do developers struggle with cryptography APIs? in Proceedings of the 9th International Conference on Software Engineering, ser. ICSE’16, 2016. [Online]. Available: http://doi.acm.org/10.1145/2884781.2884790

[7] M. Georgiev, S. Iyengar, S. Jana, K. 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, 2012, pp. 38–49. [Online]. Available: http://doi.acm.org/10.1145/2382196.2382204

[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, 2009, pp. 320–330. [Online]. Available: http://doi.acm.org/10.1145/1518365.1518422

[9] Y. Acar, M. Backes, S. Fahí, 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, 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 European Software Engineering Conference Held Jointly with International Symposium on Foundations of Software Engineering, ser. ESEC/FSE-13. ACM, 2005, pp. 306–315. [Online]. Available: http://doi.acm.org/10.1145/1081706.1081755

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

[13] A. Wasylkowski, A. Zeller, and C. Lindig, “Detecting object usage anomalies,” in Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, ser. ESEC/FSE ’07. New York, NY, USA: ACM, 2007, pp. 35–44. [Online]. Available: http://doi.acm.org/10.1145/1287624.1287632

[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. New York, NY, USA: ACM, 2007, pp. 123–134. [Online]. Available: http://doi.acm.org/10.1145/1250734.1250749

[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 Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, ser. ESEC/FSE ’09. New York, NY, USA: ACM, 2009, pp. 383–392. [Online]. Available: http://doi.acm.org/10.1145/1595609.1595676

[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. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 370–384. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-00593-0_25

[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. Washington, DC, USA: IEEE Computer Society, 2009, pp. 496–506. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2009.5070548
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.