Investigating Next Steps in Static API-Misuse Detection

Sven Amann[*] Hoan Anh Nguyen[†] Sarah Nadi[‡] Tien N. Nguyen[§] Mira Mezini[¶]

[*]CQSE GmbH [†]Amazon.com, Inc [‡]University of Alberta [§]University of Texas-Dallas [¶]Technische Universität Darmstadt

research@sven-amann.de hoanamzn@amazon.com nadi@ualberta.ca tien.n.nguyen@utdallas.edu mezini@cs.tu-darmstadt.de

Abstract—**Application Programming Interfaces (APIs) often impose constraints such as call order or preconditions. API misuses, i.e., usages violating these constraints, may cause software crashes, data-loss, and vulnerabilities. Researchers developed several approaches to detect API misuses, typically still resulting in low recall and precision. In this work, we investigate ways to improve API-misuse detection. We design MuDetect, an API-misuse detector that builds on the strengths of existing detectors and tries to mitigate their weaknesses. MuDetect uses a new graph representation of API usages that captures different types of API misuses and a systematically designed ranking strategy that effectively improves precision. Evaluation shows that MuDetect identifies real-world API misuses with twice the recall of previous detectors and 2.5x higher precision. It even achieves almost 4x higher precision and recall, when mining patterns across projects, rather than from only the target project.**

I. Introduction

Incorrect usages of an Application Programming Interface (API), or *API misuses*, are violations of (implicit) *usage constraints* of the API. An example of a usage constraint is having to check that `hasNext()` returns `true` before calling `next()` on an `Iterator`, in order to avoid a `NoSuchElementException` at runtime. API misuse is a prevalent cause of software bugs, crashes, and vulnerabilities [1]–[7].

To mitigate API misuse, researchers have proposed several *API-misuse detectors* [1], [8]–[17]. These detectors analyze *API usages*, i.e., code snippets that use a given API. The detectors commonly mine *usage patterns*, i.e., equivalent API usages that occur frequently, and then report deviations from these patterns as potential misuses. Unfortunately, the reported precision of such detectors is typically low and a recent study [18] showed that their recall is also very low. Thus, we need better detectors to address the still-prevalent problem of API misuse [19], [20].

Previous work identified individual as well as common strengths and weaknesses of existing detectors [18] in an empirical study using the open-source benchmark MuBench [21]. In this paper, we investigate whether addressing the reported weaknesses indeed leads to better performance in practice. Therefore, we design a new misuse detector, MuDetect. MuDetect encodes API usages as API-Usage Graphs (AUGs), a comprehensive usage representation that captures different types of API misuses. MuDetect employs a greedy, frequent-subgraph-mining algorithm to mine patterns and a specialized graph-matching strategy to identify pattern violations. Both components consider code semantics to improve the overall detection capabilities. On top, MuDetect uses an empirically optimized ranking strategy to effectively rank true positives. While previous detectors mostly target a per-project setting [18], MuDetect also works in a cross-project setting, where it mines thousands of usage examples from third-party projects.

We assess the precision and recall of MuDetect and show that it outperforms the four state-of-the-art detectors evaluated in prior work [18]. In our evaluation, we extended MuBench by 107 real-world misuses identified in a recent study on run-time verification [19]—more than doubling its size—to ensure that our design decisions generalize. We show that, in a setting with perfect training data, MuDetect achieves a recall of 72.5%, which is 20.3% higher than the next best detector and over 50% higher than the other detectors. In the typical per-project setting, MuDetect achieves recall of 20.9%, which is 10.2% better than the second-best detector, and precision of 21.9%, which is 13.1% better than the second-best detector. In a cross-project setting, MuDetect's recall and precision again improve significantly to 42.2% and 33.0%, respectively. Throughout the experiments, MuDetect identified 27 previously unknown misuses, which we reported in eight pull requests (PRs). To date, three of the PRs got accepted, demonstrating that MuDetect identifies actual issues in current software projects.

To summarize, this paper makes the following contributions:

- AUG, a graph-based representation of API usages that captures all usage properties relevant for identifying misuses.
- Code-semantic-aware, greedy frequent-subgraph-mining and graph-matching algorithms to identify patterns within and across projects and (violating) instances in a target codebase.
- MuDetect, a (cross-project) misuse detector.
- An empirical study of ranking strategies to improve precision.
- An empirical evaluation that compares MuDetect to existing detectors, and includes an analysis of the results to identify further opportunities for improvement.
- Fixes for all previously unknown misuses identified by MuDetect, for external validation of the findings' relevance.

We publish our MuBench extension, MuDetect's implementation, and all experiment data, tooling, and results [22].

II. State of the Art and Improvement Strategies

Our work focuses on static API-misuse detection in Java. In the following, we first briefly introduce the state-of-the-art detectors that were empirically evaluated by Amann *et al.* [18]. Subsequently, we summarize the problems that their study revealed with these detectors and outline how we mitigate them. The detectors work as follows.

GROUMINER [12] represents usages as directed acyclic graphs that encode method calls, field accesses, and control structures as nodes and control-/data-flow dependencies among them as unlabelled edges. GROUMINER uses sub-graph mining to find patterns and then detects violations of these patterns as missing nodes. It detects missing method calls and misplaced method calls, as well as missing control structures.

JADET [10] encodes the transitive closure of the call-order relation in each usage as pairs of the form `m()≺n()`. It uses Formal Concept Analysis [23] to identify violations, i.e., rarely missing pairs. It cannot detect violations of patterns with only one pair. TIKANGA [16] builds on the same algorithm, but encodes usages using temporal properties (CTL). Both detectors detect missing and misplaced calls.

DMMC [1] encodes usages as sets of methods called on the same receiver type. It identifies violations by computing, for every usage, the ratio of the number of equal usages and usages with one additional call. Intuitively, a violation should have few exactly-similar usages, but many almost-similar usages.

The problems that Amann *et al.* [18] identified as the root causes for the low recall and precision of these detectors as well as our strategies to mitigate them are as follows.

P1: **Representation.** On average, 45.8% of false negatives were due to the inability of the detectors' underlying representations to capture details necessary for differentiating misuses from correct usages [18]. For example, DMMC and GROUMINER encode methods by their names only and, hence, cannot detect a missing method call when an overloaded version of the method is called (e.g., the misuse calls `String.getBytes()` while the pattern requires `String.getBytes(String)`). *Our representation tracks method-call arguments. Additionally, for the first time, we provide a representation that combines tracking of control flow, exceptional flow, order of method calls, synchronization, and data flow.* Previous detectors considered these features in isolation.

P2: **Matching.** On average, 31.3% of false negatives were due to detectors not matching patterns to misuses [18]. For example, to identify that two methods are called in the wrong order, say `b();a()` instead of `a();b()`, a detector needs to both capture the call order, and match the pattern and misuse despite the different order. Similarly, a detector needs to consider sub-typing information to match a `Collections.size()` call found in a pattern to an `ArrayList.size()` call found in a usage. Another issue is that some detectors use a distance threshold to filter their findings, which may filter true positives, e.g., if a misuse contains additional, optional method calls. MUDETECT *matches calls even if their order differs, considers type-hierarchy information, and does not employ a distance threshold, but rather enforces a ranking strategy.*

P3: **Uncommon Usages.** On average, 34.3% of the false positives were uncommon-but-correct usages [18]. It is generally difficult, if not impossible, to automatically and precisely distinguish uncommon usages from misuses. However, the study observed that many of the false positives for uncommon usages involved methods without side effects, *pure methods,*

such as getters. Since invocations of pure methods cannot be required, unless their return value is actually needed, MUDETECT *removes calls to pure methods from patterns, unless their return value is used in the pattern.*

P4: **Alternative Patterns.** On average, 19% of false positives were usages that violate some particular pattern, but conformed to another (alternative) pattern [18]. Such *alternative-pattern instances* may even accumulate to 28% of the false positives [15]. Therefore, MUDETECT *filters alternative-pattern instances.*

P5: **Self- and Cross-method Usages.** On average, 12.2% of false positives were due to detectors not distinguishing self- and cross-method usages [18]. In a *self usage*, a class uses part of its own API in its implementation, e.g., `Collection.addAll()` calls `Collection.add()`. Constraints that client usages must adhere to, e.g., guarding calls by checks, may not apply to self usages. In a *cross-method usage*, an object is used by multiple methods, e.g., by storing it in a field. From the perspective of an individual method, we have only a partial view on the entire usage scattered across methods. For both types of usages, an intra-procedural analysis potentially detects partial usages, i.e., violations, that are not actual misuses. Therefore, MUDETECT *ignores self-usages and usages on fields.*

P6: **Ranking.** Often, the detectors correctly identified misuses, but ranked them extremely low [18]. An effective ranking mechanism that pushes true positives to the top is essential for saving developers' efforts. *We empirically investigate several ranking factors from the literature and compose a ranking strategy that effectively prefers true positives.*

P7: **Usage Examples.** A possible cause of the detectors' low recall is a lack of correct usages in their training projects [18]. To validate this hypothesis, *we evaluate* MUDETECT *in both a per-project setting, which is the norm in the literature, and a cross-project setting that provides more training examples.*

III. MUDETECT

We design a new API-misuse detector, MUDETECT, that adopts the strengths of previous detectors and addresses the problems summarized in Section II as follows:

1) We design API-Usage Graphs (AUGs), a representation of API usages that simultaneously captures many properties that can distinguish misuses from correct usages.
2) We design a new pattern-mining algorithm, based on frequent-subgraph mining, that exploits domain knowledge about API usages to efficiently identify usage patterns.
3) We design a new detection algorithm, which uses domain knowledge to efficiently identify API-usage violations.
4) We design a ranking strategy that effectively ranks true positives before false positives.

A. API-Usage Graphs

Amann *et al.* [18] found the graph-based GROUM representation of usages to be most promising for identifying misuses. However, GROUMs still capture insufficient details (*P1* (Representation)), which is why we propose API-Usage Graphs (AUGs) as a new representation of API usages. An
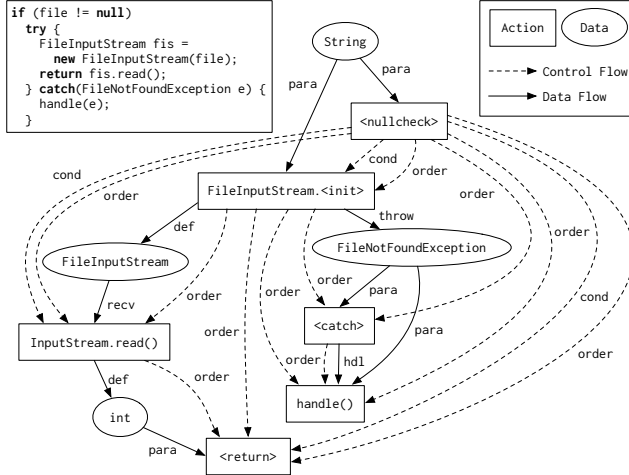
Figure 1: An API Usage and its API-Usage Graph.

AUG is a directed, connected multigraph with labelled nodes and edges. Nodes represent data entities, such as variables, and actions, such as method calls; edges represent control and data flow between entities and actions represented by nodes. Figure 1 shows an example. MuDETECT's intra-procedural analysis creates one AUG from each source method.

1) Usage Actions: We use *action nodes* to represent method calls, operators, and instructions in API usages (boxes in Figure 1). For method calls, we use labels `T.M()`, where `M` is the method's name and `T` is the simple name of its declaring type. For constructor calls, we use labels of the form `T.<init>`. Using the declaring type abstracts over different static receiver types (*P2* (Matching)); e.g., all calls to `size()` on a `List`, `LinkedList`, or `ArrayList` are labelled `Collection.size()`.

We encode equality and relational operators to capture conditions such as `list.size()>0`. To abstract over alternative ways to express a condition, e.g., `l.size()!=0` and `!(l.size()==0)`, we use the label `<r>` for all equality and relational operators and drop negation operators. To also abstract over alternative ways to compose conditions, e.g., `a&&b` and `!(!a||!b)`, we drop the conditional operators `&&` and `||`. With this abstraction level, we focus on detecting the absence or presence of conditions in API usages, rather than logical mistakes in conditions. We capture `null` checks, e.g., the `null` check on `file` in Figure 1, by action nodes with the dedicated label `<nullcheck>` to distinguish this special condition from other comparisons. We encode unconditional control instructions, such as `return`, `throw`, and `catch`, by action nodes with dedicated labels, e.g., the `<catch>` and `<return>` nodes in Figure 1.

To reduce false positives due to *P5* (Self- and Cross-method Usages), we heuristically exclude self- and cross-method usages from AUGs: We create no nodes for method calls on `this` and `super` as well as on field accesses on both these qualifiers.

2) Data Entities: We use *data nodes* to represent objects, values, and literals that appear in API usages (ovals in Figure 1). We encode data entities as nodes to make data dependencies between actions, such as multiple calls on the same object, explicit, to ensure we have a connected subgraph with all data-dependent parts of a usage, and to distinguish overloaded versions of methods by their parameter entities. We uniformly create data nodes for variables, fields, and objects that are not assigned but immediately used, e.g., in a method-call chain.

Since certain types, such as `List`, `ArrayList`, and `LinkedList`, appear almost interchangeably in API usages, we label all data nodes `<Object>`. This allows us to abstract over different static types (*P2* (Matching)), while checking the data-/control-flow that the data entities take part in. Note that Figure 1 shows the simple type names for better readability.

3) Control Flow and Data Flow: We use edges to represent control flow and data flow. We distinguish eight types of edges and label them with their type. Figure 1 shows seven of these edge types, labelled with acronyms for brevity.

- A *receiver* edge connects from a data node to a method call that is invoked on the respective object.
- A *parameter* edge connects from a data node to an action that takes the respective object or value as a parameter.
- A *definition* edge connects from an action that creates or returns a value or object to the respective data node.
- An *order* edge connects, in order of execution, two action nodes operating on the same data entity (receiver or parameter). Since we want MuDETECT to discover wrong method-call order, we over-approximate temporal relations by building the transitive closure over *order* edges. To keep AUGs acyclic, we exclude backwards edges from loops.
- A *condition* edge connects an action whose result controls branching to an action controlled by that branching.
- A *synchronize* edge connects a data node that the program obtains a lock on to an action executed under that lock.
- A *throw* edge connects an action that may throw an exception to a data node representing that exception object. We use the `throws` information, if it is resolvable, to determine which exception may be thrown by an action. We connect exception data nodes to respective `<catch>` nodes with parameter edges.
- A *handle* edge connects from a `<catch>` node to an action in a respective exception handling block.

This detailed dependency information helps distinguish misuses from correct usages (*P1* (Representation)), relate usages despite notational differences (*P2* (Matching)), and consider code semantics in both pattern mining and violation detection.

B. Pattern Mining

Listing 1 shows our pattern-mining algorithm, which takes a set A of AUGs, a frequency measure f and a frequency threshold σ, and produces a set of patterns. A *pattern* is a sub-AUG that occurs frequently in A, and a *pattern instance* is an occurrence. A sub-AUG p is a pattern if it has $f(p) \geq \sigma$ instances. The algorithm follows three key ideas:

1) Apriori-based Mining: The algorithm follows the general idea of an apriori-based algorithm for frequent-subgraph mining [24], i.e., it mines patterns by starting from all single-method-call patterns (Line 2) and recursively extending them to larger patterns (Line 3). The key idea here is that if a graph occurs frequently, all of its subgraphs also occur frequently. To extend a pattern p of size k, the algorithm generates all suitable

```
1  def mine(A: Set[AUG], f: Pattern → int, σ: int)
2    P₀ = {p | p ∈ single_call_patterns(A) and f(p) ≥ σ}, P = ∅
3    for p in P₀: extend(p, P, f, σ)
4    return P
5
6  def extend(p: Pattern, P: Set[Pattern],
7      f: Pattern → int, σ: int)
8    E = {e | i ∈ p and e ∈ generate_extensions(i)}
9    PC = {c | c ∈ isomorphic_clusters(E) and f(c) ≥ σ}
10   UC = PC \ P
11   if UC ≠ ∅:
12     c = most_frequent(UC)
13     extend(c, P, σ)
14   else: P = P ∪ {p}
15   ip = {i | i ∈ p and ∀c ∈ PC. generate_extensions(i) ∩ c = ∅}
16   if f(ip) ≥ σ: P = P ∪ {ip}
17
18 def generate_extensions(i: Instance)
19   extensions = ∅
20   for n in adjacent_nodes(i):
21     if has_non-order_connection(n, i) and (
22        is_non-pure_call(n)
23        or (is_pure_call(n) and has_out_connection(n, i))
24        or (is_operator(n) and has_in&out_connection(n, i))
25        or (is_data(n) and has_out_connection(n, i))):
26       extensions = extensions ∪ {i ⊕ n}
27   return extensions
```

Listing 1: MUDETECT's Pattern-Mining Algorithm

extensions of size $k + 1$ for all pattern instances of p (Line 8). This is done by exploring each adjacent node (Line 20). When i is extended by an adjacent node, all edges from a that connect i and the node are added as well. Extending by nodes, as opposed to by edges, enables scalable mining of AUGs, which usually have a large number of edges.

2) Code Semantics: When extending a pattern instance i, the algorithm distinguishes different types of adjacent nodes. Specifically, it decides whether an adjacent node is suitable for extending i as follows: A node that is only connected by an order edge is unsuitable (Line 22). Otherwise, a non-pure method call is always suitable (Line 22). A pure method call is suitable only if it has an outgoing edge to a node in i (Line 23), i.e., if it defines a data node or controls an action node in i. Since pure methods have no side effects, they can impact a usage only through their return value. To avoid the complexity of inter-procedural analysis, the algorithm identifies pure methods heuristically: It considers any method whose name starts with get as pure, since getters are mostly pure and very prevalent. An operator is suitable only if it has at least one incoming and one outgoing edge to i (Line 24), because operators are like pure methods whose result is based solely on their parameters, as opposed to parameters and state. A data node is suitable only if it has an outgoing edge to i (Line 25), i.e., when it is used in the usage. These decisions based on code semantics contribute to obtaining meaningful patterns, thereby mitigating the problem of flagging uncommon usages as misuses (*P3 (Uncommon Usages)*).

3) Greedy Exploration: To identify $(k + 1)$-patterns in the set of all extensions of the instances of p, the algorithm clusters isomorphic extensions to pattern candidates (Line 9). To reduce the complexity of graph isomorphism detection, the algorithm uses a heuristic that combines graph vectorization and hashing [25]. More specifically, a graph is represented as a vector of features, each of which is extracted from the

labels of a sequence of nodes and edges along a path in the graph. Two graphs are isomorphic if their corresponding feature vectors have the same hash value. The algorithm then filters out all candidates that it found before (Line 10). If there are no further frequent extensions of p, i.e., p is inextensible, the pattern is added to the set of final patterns P (Line 14). If any unexplored candidate remains (Line 11), the algorithm selects the most-frequent one (Line 12) and recursively searches for larger patterns (Line 13). This greedy strategy avoids the combinatorial explosion problem of exhaustive search with backtracking and makes our mining scale to a large number of large graphs, unlike GROUMINER, which often timed out [18].

In addition to the possible extensions, the algorithm also keeps track of those instances that do not have any frequent extension (Line 15). If these inextensible pattern instances are themselves frequent, it adds this pattern to P (Line 16). The intuition is that an API might have a core pattern and additional alternative patterns that contain it (*P4 (Alternative Patterns)*).

C. Violation Detection

Listing 2 shows our detection algorithm. It takes a set T of target AUGs, a set P of patterns, and a ranking function r and produces a list of violations. A *violation* is a strict subgraph of a pattern. The algorithm consists of four major steps:

1) Graph Matching: The detection algorithm first checks each pair of a target and a pattern for common subgraphs (Line 6). To identify the subgraphs, the algorithm follows the general idea of the pattern-growth approach for frequent-subgraph mining [24], i.e., it discovers the largest common subgraphs of each pair of a pattern and a target (Line 6), by starting from all common method-call nodes (Line 16) and recursively extending the common subgraph (Line 17), one adjacent edge at a time. This allows us to find even single missing edges, e.g., wrong order of two method calls.

When searching for possible mappings of a pattern AUG onto a target AUG, the detection algorithm follows a greedy extension strategy. It continuously selects the next-best pattern edge, while exploring all alternative mappings to the target. This avoids the combinatorial explosion problem of an exhaustive search with backtracking. The algorithm explores all alternatives in the target, as opposed to in the pattern, because targets are usually larger and, therefore, likely contain more alternatives. This results in higher precision.

When exploring candidates, there may be multiple equivalent candidate extension edges. Two edges are *equivalent* if they have the same type, both their source and target nodes have the same label, respectively, and mapping them onto each other is consistent with the current mapping between target and pattern nodes. The node mapping is *consistent* if every node from the target is mapped to at most one node from the pattern and vice versa. Intuitively, the more equivalent edges, the more alternative mappings exist and the more likely it is to select a non-optimal mapping. To decrease this likelihood, the algorithm counts equivalent edges in the target and the pattern (Line 26) and gives priority to edges with fewer equivalent alternatives.

```
1  def find_violations(T: Set[AUG], P: Set[Pattern],
2    r: (Set[Violation], Set[Instance], Set[Pattern]) →
         List[Violation]):
3    V = ∅, I = ∅
4    for target in T:
5      for pattern in P:
6        for overlap in common_subgraphs(target, pattern):
7          if overlap = pattern:
8            I = I ∪ {Instance(target, pattern)}
9          elif overlap ⊏ pattern:
10           V = V ∪ {Violation(target, pattern, overlap)}
11   V_A = filter_alternatives(V, I)
12   V_R = r(V_A, I, P)
13   return filter_alternative_violations(V_R)
14
15 def common_subgraphs(t: AUG, p: Pattern):
16   S = single_call_node_overlaps(t, p)
17   return {lcs | lcs ∈ extend_subgraph(s, t, p) and s ∈ S}
18
19 def extend_subgraph(o: Overlap, t: AUG, p: Pattern)
20   e = next_extension_edge(o, t, p)
21   return extend_subgraph((o ⊕ e), t, p) if e ≠ none else o
22
23 def next_extension_edge(o: Overlap, t: AUG, p: Pattern):
24   e_best = none, w_min = inf
25   for e in adjacent_edges(o, t):
26     w = count_equiv_edges(e, t) * count_equiv_edges(e, p)
27     if 0 < w and w < w_min:
28       e_best = e, w_min = w
29   return e_best
```

Listing 2: MUDETECT's Detection Algorithm

Mapping these first eliminates equivalent alternatives that are inconsistent with the extended node mapping.

2) Alternative-Pattern Instances: There may be alternative ways to use an API, e.g., before fetching an item from a Set, we may either check that it is !empty() or that it has size() > 0. If we have patterns for both cases, these overlap, since fetching an item requires the same calls in both cases. Consequently, an instance of one of the patterns violates the other pattern (*P4* (Alternative Patterns)), because the instance shares elements with both patterns and either misses the size or the emptiness check. Following this insight, our detection algorithm sorts each common subgraph of a target and a pattern into one of two categories: pattern instances, i.e., subgraphs equal to the pattern (Line 7), and violations, i.e., strict subgraphs of the pattern (Line 9). Once all targets and patterns are processed, it uses the set of instances to filter out violations that are subgraphs of instances of another pattern (Line 11).

3) Violation Ranking: After identifying all violations in the target code base, the detection algorithm ranks the findings (Line 12). Section III-D discusses ranking strategies in detail.

4) Alternative Violations: If a usage violates all alternative patterns, the filtering for alternative-pattern instances (Section III-C2) leaves all respective violations in place. To avoid such duplicates, we filter violations involving a method call that is also part of a violation at a higher rank (Line 13).

D. Ranking

Ranking the detected violations is crucial for MUDETECT's precision, since it controls how many true positives appear among the top findings (*P6* (Ranking)). The ranking may also impact MUDETECT's recall, since we filter alternative violations based on the ranking order of the findings (see Section III-C4), which may eliminate true positives. To design MUDETECT's

ranking strategy, we first survey existing ranking strategies and discuss their individual factors. Then, we compose new ranking strategies from these factors.

a) Previous Ranking Strategies: Some detectors use a maximal distance between a pattern and a usage to classify the usage as a violation [10], [12], [16], where *distance* is the number of facts from the pattern that the usage misses. *Facts* might be method calls, order relations between call pairs, or nodes and edges, depending on the usage representation. Intuitively, usages that are distant from a pattern P are more likely occurrences of an alternative pattern than violations of P. We compute the distance between a pattern and a usage AUG via the number of nodes and edges n_m that the usage misses from the pattern. We normalize n_m by the total number of elements n_p of the pattern. Since a missing node always implies that all edges connecting to it are also missing, we take the number of missing edges from/to missing nodes n_e out of the equation. This leads to our *violation-overlap measure* $v_o = (n_m - n_e)/(n_p - n_e)$.

Some detectors rank their findings by the support of the violated patterns (p_s) [8], [14], [15]. Intuitively, p_s expresses the miner's certainty regarding the correctness of the pattern.

Monperrus *et al.* [1] rank their findings by the *confidence*, combining p_s and the *number of violations of the pattern (p_v)* into $p_s/(p_s + p_v)$. Intuitively, patterns with more violations more likely contain usage properties that are not mandatory, making their violations more likely to be false positives.

Some detectors [8], [12] rank their findings by their *rareness*, combining p_s and the number of times the violation reoccurs, i.e., the *violation support (v_s)*, into $(p_s - v_s)/p_s$. Intuitively, a violation that occurs more often is less likely to be problematic.

Wasylkowski *et al.* [10] rank findings by a *defect indicator*, combining p_s, v_s, and a *pattern-uniqueness factor (p_u)*, into $p_u \times p_s/v_s$. To compute p_u, they count for every API in the pattern the number of violations involving that API and take the inverse of the largest such number. Intuitively, if an API is involved in more violations, any particular violation involving it is less likely to be problematic.

b) MUDETECT's Ranking Strategy: As candidates for our ranking strategy, we consider the strategies from the literature and all combinations of the individual ranking factors by multiplication. For the latter, we use p_s, p_u, and v_o as is, but invert p_v and v_s, such that smaller values imply lower probability of the violation being problematic. We multiply them, such that, if any of the factors is low, the overall ranking weight is low. Since it is unclear which candidate is most useful, we empirically evaluate them. We explain the respective experiment in Section IV-B and its results in Section V-A.

E. Per-project and Cross-project Settings

As Sections III-B and III-C show, MUDETECT separates pattern mining and detection, which allows us to run it in two different settings. The first is a *per-project setting*, where we configure MUDETECT to use the AUGs from its target project as the input for both pattern mining and violation detection. This enables a fair comparison to existing detectors, which combine

mining and detection in a single phase [18] and, thus, always mine and detect on the same input. In this setting, we follow existing work [8], [10], [12], [16] and define the frequency measure $f(p)$ as the number of distinct instances of the pattern.

The second is a *cross-project setting*, where we configure MUDETECT to use the AUGs from its target project as the input for violation detection and AUGs from other projects as input for pattern mining. This allows us to provide additional usage examples for mining (*P7 (Usage Examples)*). We call this configuration MUDETECTXP. In this setting, we define the frequency measure $f(p)$ as the number of projects from which at least one instance of the pattern originates. The intuition is that a pattern that occurs in more projects is a generally reusable pattern and, therefore, more likely to be correct than a pattern that occurs only in a single project (a project-specific pattern), even if it occurs frequently within that project.

IV. EVALUATION SETUP

We now present the setup that we use to compare MUDE-TECT's precision and recall to existing detectors. We aim to understand the effectiveness of our mitigation strategies and the impact of the ranking strategies discussed in Section III-D.

A. Detectors and Dataset

We compare MUDETECT against the four detectors JADET, GROUMINER, TIKANGA, and DMMC, which were empirically evaluated by Amann *et al.* [18]. As the ground-truth for the experiments, they used MUBENCH [7], a dataset of open source projects with 84 known API misuses (Table I, Row 1). For 64 of these misuses, MUBENCH also contains examples of correct usages, which are derived from the fix of the misuse. Since we designed MUDETECT using insights from Amann *et al.*'s study, an evaluation only on MUBENCH may suffer from overfitting. Therefore, we extend the dataset by misuses identified in a recent study by Legunsen *et al.* [19]. They applied runtime verification of API specifications to 200 open-source projects and submitted 114 pull requests that fix API misuses identified in this process. From this set, we take all misuses for which the pull request was accepted as of August 8, 2017, which adds 107 new misuses from 30 projects to our experiments (Table I, Row 2).[1] Following the structure of MUBENCH, we derive examples of correct usage from the accepted pull requests.

Overall, this gives us a benchmark dataset with 191 API misuses from real-world projects (Table I, Row 3). We use this dataset in our experiments. For simplicity, we refer to this extended dataset as MUBENCH throughout the rest of the paper.

B. Experimental Setup

To evaluate MUDETECT, we conduct the three per-project experiments proposed by Amann *et al.* [18]: Experiment P to measure precision, Experiment RUB to determine recall upper bound, and Experiment R to measure actual recall. We also

---

[1]JADET and TIKANGA initially crashed on most new projects, because they used the outdated Bytecode toolkit ASM 3.1. To fix this, we updated this dependency to ASM 6.0. To ensure that this change did not hamper with the detectors' capabilities, we repeated the experiments of Amann *et al.* [18] and verified that the detectors still produce the exact same results.

Table I: MUBENCH: Number of Misuses (#MU) and Number of Misuses with Corresponding Correct Usages (#CU).

Dataset	#MU	#CU
Original MUBENCH [18]	84	64
Our Extension	107	107
Extended MUBENCH	191	171

Table II: Experiment RNK: Number of Hits (#H), Average Hit Rank (AHR), and Number of Hits in the Top-20 (@20).

# Strategy	@20	#H	AHR	# Strategy	@20	#H	AHR
1. $p_s/v_s \times v_o$	19	34	91.6	...			
2. p_s/v_s	17	34	91.8	9. p_s	14	34	305.5
3. $p_s/v_s \times v_o \times p_v$	16	34	90.1	...			
4. Rareness	16	33	94.3	33. p_u/v_s	2	18	53.2
...				34. v_o	1	26	1187.4

design Experiment RNK to compare the ranking strategies discussed in Section III-D and Experiment XP to evaluate MUDETECTXP in a cross-project setting. We set the frequency threshold $\sigma = 10$ for MUDETECT and $\sigma = 5$ for MUDETECTXP. For the other detectors, we use their best configurations from the respective publications.

We execute the experiments using MUBENCHPIPE [18], a public automated benchmarking pipeline built on top of MUBENCH. MUBENCHPIPE facilitates preparing the target projects from MUBENCH, executing the detectors, and collecting result statistics after we manually reviewed the detectors' findings. In all our experiments, two authors first independently reviewed each detector finding and then discussed any disagreements until a consensus was reached about whether the finding correctly identifies a misuse. We report Cohen's Kappa score as a measure of the reviewers' initial agreement. We now introduce these five experiments in detail.

Experiment P. The goal of Experiment P is to measure the detectors' *precision*. We run the detectors on all projects from MUBENCH, letting them *mine patterns and detect violations* on a per-project basis. Since some detectors report several hundreds of findings, reviewing all findings of all detectors on all projects is practically infeasible. Therefore, we sample ten projects and review the top-20 findings per detector on each of them, as determined by the detectors' own ranking strategies. In this sample, we include the five projects Amann *et al.* [18] used in their precision experiment. In addition, we choose another five of the new projects we added to MUBENCH. To this end, we compute the average normalized number of findings (ANNF) across detectors for each project. The NNF for a given detector for a given project is the number of findings the detector has on that project divided by the maximum number of findings the detector has on any project. We select two projects with the highest ANNF, two projects with the lowest ANNF, and one random project from the mid range. For fairness, we exclude projects where one of the detectors failed or where two or more detectors did not report any findings. We could not exclude all projects where one of the detectors did not report any findings, because this left us with fewer than five projects to chose from.

Experiment RUB. The goal of Experiment RUB is to assess

detectors' *recall upper bound*, given perfect training data. This separates conceptual limitations from the effect of insufficient training data. Since we need to provide a correct usage as input training data, we limit this experiment to the 171 misuses in MUBENCH that have corresponding correct usages (see Table I). We run the detectors once for each of these misuses, providing them with enough copies of the corresponding correct usage for pattern mining. This ensures that detectors always find sufficient evidence to mine the pattern required to identify the misuse. We review all *potential hits*, i.e., all detector findings in the same method as the known misuse.

Experiment R. The goal of Experiment R is to measure the detectors' *recall*. We run the detectors on all projects of MUBENCH, letting them *mine patterns and detect violations* on a per-project basis. Then, we again review all potential hits. As the ground truth, we use all 191 known misuses from MUBENCH, plus any previously unknown true positives identified by the detectors in Experiment P. This gives us the recall of the detectors with respect to a large number of misuses and, at the same time, crosschecks which of the detectors' findings are also identified by other detectors.

Experiment RNK. The goal of Experiment RNK is to find *the best ranking strategy* for MUDETECT among the candidate strategies discussed in Section III-D. Ideally, we would repeat both Experiments P and R for all 34 candidate ranking strategies to determine the best strategy. However, repeating Experiment P would require us to review up to 20 findings for each of the ten target projects per candidate strategy – a total of 6,800 findings – , which is practically infeasible. Therefore, we only repeat Experiment R for each of the candidate ranking strategies. This gives us both the recall of the detector, as well as the ranks of all confirmed hits, i.e., findings that identify a known misuse from MUBENCH. We use the *number of hits*, the *average rank of all hits*, and the *number of hits in the top-20 findings* as quality measures for the ranking strategies. To prevent overfitting, we use only the original MUBENCH for Experiment RNK and verify the result on our extended dataset.

Experiment XP. The goal of Experiment XP is to measure MUDETECTXP's *precision* and *recall* (in a cross-project setting). To measure precision, we run MUDETECTXP on the ten sample projects from Experiment P and review its top-20 findings. To measure recall, we run MUDETECTXP on all projects in MUBENCH and review all its potential hits for all known misuses, as in Experiment R. For each target project, we provide the detector with training projects for all APIs with known misuses in the target project. To ensure that the training projects contain examples of the APIs with known misuses in MUBENCH, we collect client projects of the respective APIs using the code-repository mining platform BOA [26] (full 2015 GitHub dataset). For each API, we query BOA for projects that either declare a field, variable, or parameter, or call a static method of the respective API type. We share the query template and the result lists [22]. From each list, we take the first 50 projects that are still available as of February 2018 and randomly sample up to 20 usage examples of the respective API from each project.

This gives us a diverse cross-project sample of up to 1,000 usage examples per API.

V. RESULTS

In this section, we present the results of our experiments and compare MUDETECT's detection performance with the detectors JADET, GROUMINER, TIKANGA, and DMMC. All experiments ran on a *MacBook Pro* with an *Intel Xeon @ 3.00GHz* and *32GB of RAM*. Tables II and III summarize the results; the full results are available on our artifact page [22].

We analyze the root causes for MUDETECT's false negatives (FN) and false positives (FP) in each subsection, as applicable, to validate whether our mitigation strategies were successful and to direct future work. We present the most prevalent root causes here and the full list on our artifact page [22]. We also discuss possible mitigation strategies and their trade-offs.

A. Experiment RNK

We first run Experiment RNK on the original MUBENCH to determine the best ranking strategy for MUDETECT. Table II shows the best and worst ranking strategies ordered by the number of hits in the top-20 findings (@20), the number of hits (#H), and the average hit rank (AHR).

The results show that ranking has a huge impact on how high MUDETECT ranks the misuses. We observe that the pattern support (p_s) appears in all of the top-16 and in none of the bottom-10 ranking strategies. Contrarily, the pattern uniqueness (p_u) appears in 11 of the bottom-15 strategies and in none of the top-10. The violation-overlap measure (v_o), the violation support (v_s), and the pattern violations (p_v) appear in different combinations in ranking strategies throughout the field. While this clearly shows that the pattern support is the most important ranking factor, the strategy consisting of only this factor is only the 9th-best strategy. This suggests that detectors should consider other factors as well. The best strategy combines the pattern support (p_s), the support of the violations (v_s), and the violation-overlap measure (v_o) into $p_s/v_s \times v_o$. Repeating Experiment RNK on our dataset extension identifies the same best ranking strategy. We use this strategy for MUDETECT and MUDETECTXP in all remaining experiments.

B. Experiment P

The first part of Table III summarizes the results of measuring the detectors' precision in their top-20 findings.

> *O1:* MUDETECT reports 146 violations in the top-20 findings in the ten projects. Among these violations, we find 32 true positives, 21 of which were previously unknown. This results in precision of 21.9%, which exceeds the precision of the other detectors more than two-fold.

The results of Experiment P show that the ranking strategy identified from Experiment RNK successfully pushes true positives to the top (*P6* (Ranking)), allowing us to outperform other detectors. MUDETECT also reports no false positives that are instances of alternative patterns. Since such false positives accumulate to 19% of other detectors' findings [18],

we conclude that our filtering strategy successfully resolves *P4* (Alternative Patterns). MuDetect reports no self-usages and only one cross-method usage, which our filtering misses, because the respective object is initialized as a local variable and only later assigned to a field. While this is an opportunity to improve our filtering heuristic, it also shows that our strategy successfully mitigates *P5* (Self- and Cross-method Usages), which caused 12% of the false positives of other detectors [18]. However, there are still false positives, due to different causes:

***FP1:* Uncommon Usages.** 84 (73.7%) of the false positives are uncommon-but-correct usages. In nine cases, e.g., a loop calls `Iterator.hasNext()` again after calling `next()`, to check whether there will be a subsequent iteration. MuDetect reports a missing call to `next()` after this second call to `hasNext()`. This illustrates two problems: (1) the heuristic for identifying pure methods by the name prefix `get` misses cases such as `hasNext()`, (2) MuDetect does not consider alternative non-frequent patterns. This root cause of false positives corresponds to *P3* (Uncommon Usages). We conclude that the removal of calls to pure methods is insufficient to address this problem. A future solution might be *a probabilistic model of API usage* that considers the likelihood of different usages and reports no violation if one usage is only slightly more likely than another, or if an API's usages generally vary a lot.

***FP2:* Intra-procedural Analysis.** 18 (15.8%) of the false positives are due to our intra-procedural analysis. In seven cases, MuDetect reports missing usage elements that occur in transitively called methods. Using an inter-procedural analysis, e.g., to filter such false positives as proposed by Li and Zhou [8], might help mitigate this problem. Future work should investigate whether the additional computational cost pays off.

C. Experiment RUB

The second part of Table III summarizes the results of measuring the upper bound to the detectors' recall.

> *O2:* MuDetect identifies 124 of the 171 misuses used in this experiment (72.5%). This upper bound of recall clearly outranks that of the other detectors by 20.3% for GrouMiner and by over 54% for each of the other three detectors.

O2 shows that we successfully mitigated *P1* (Representation) with the design of AUGs: (1) AUGs capture the difference between correct usages and misuses better than all other detectors' representations and (2) our detection algorithm succeeds in identifying these differences. There are still 15 false negatives due to *P1* (Representation), all cases where an illegal parameter value (constants or literals) is passed as a call parameter. None of the detectors can detect these, because they do not capture concrete values.

MuDetect correctly matches pattern and target usages despite different call order and polymorphic calls, which means we successfully mitigated *P2* (Matching). There are still seven false negatives where it does not match the respective pattern and target usages, because they contain only a single, distinct call. None of the detectors identifies these cases, because they only match patterns and usages with at least one common call.

Overall, MuDetect identifies 39 misuses that all other detectors miss. In turn, MuDetect misses ten misuses that at least one of the other detectors finds, eight of which are due to the heuristics we introduced to improve precision, such as filtering cross-method usages. There are 38 more misuses that all detectors miss. False negatives of MuDetect are due to:

***FN1:* Self-Usages.** Eight cases are due to our removal of self-usages, which successfully mitigated *P5* (Self- and Cross-method Usages) in Experiment P. This means we traded recall for precision. By *capturing inter-procedural usages* we might make filtering self- and cross-method usages unnecessary and enable us to identify misuses in them. The Chronicler [11] detector mines usages from an inter-procedural call graph, which might mitigate the problem. However, it is unclear how to adapt this approach from considering only method calls to all usage elements encoded in AUGs. Furthermore, such an approach duplicates evidence, if methods are called multiple times, which might bias the mining.

***FN2:* Redundant.** Seven cases are misuses where the usage has a redundant element that should be removed. Since all detectors are designed to detect missing elements, none can detect these misuses. It is worth noting that DroidAssist [27] uses a *probabilistic approach* that might find superfluous method call, but the technique has never been evaluated.

D. Experiment R

Overall, the detectors identified 34 previously unknown misuses in Experiment P. With the 191 misuses from MuBench, this gives us 225 misuses for measuring the detectors' recall. The third part of Table III summarizes the results.

> *O3:* MuDetect identifies 47 of the 225 misuses. This results in recall of 20.9%, which exceeds the recall of the other detectors almost two-fold.

MuDetect correctly identifies 13 misuses that none of the other detectors identifies, eleven of which it already identified in Experiment P. The other six previously unknown misuses from Experiment P are also identified by at least one detector.

MuDetect misses 13 misuses that one of the other detectors finds. Six of these are identified only by DMMC, because the projects contain too few usage examples for the other detectors to mine a respective pattern. DMMC's probabilistic approach may identify misuses with little evidence. In three of the six cases, DMMC finds exactly two usage examples of the respective API: a correct usage and the misuse. Consequently, $p_s = 1$ and $p_v = 1$ and, therefore, confidence $= 0.5$, which is exactly DMMC's threshold for reporting a misuse. Since MuDetect requires a pattern support of at least 10, it cannot find these misuses. Another five of these misuses are identified by Jadet or Tikanga or both. In all cases, the target method contains multiple equal misuses. Jadet and Tikanga report a single finding identifying the misuse, but since they do not provide line locations within the method, we conservatively count it as a hit for all the misuses. MuDetect, on the other hand, reports findings at line level, which is why we only count hits when the finding line matches the known misuse

Table III: Results: Experiment P measures precision in the top-20 findings. Experiment RUB measures recall upper bound. Experiment R measures recall. Experiment XP measure precision and recall of MuDetectXP.

Detector	Experiment P			Experiment RUB			Experiment R		
	Confirmed Misuses	Precision	Kappa Score	Hits	Recall Upper Bound	Kappa Score	Hits	Recall	Kappa Score
JADET	8	8.8%	0.64	29	16.9%	0.79	15	6.7%	0.64
GROUMINER	4	2.6%	0.49	88	51.2%	0.85	7	3.1%	1.00
TIKANGA	7	8.2%	0.52	15	8.8%	0.73	17	7.6%	0.69
DMMC	12	7.5%	0.72	28	16.3%	0.88	24	10.7%	0.91
MuDetect	32	21.9%	0.90	124	72.5%	0.89	47	20.9%	1.00
MuDetectXP	30	33.0%	0.88				95	42.2%	0.93

line, resulting in only 1 hit being counted. For the last two of these misuses, MuDetect misses the pattern due to the greedy extension strategy that we chose to keep the mining scalable.

E. Experiment XP

While MuDetect has considerably higher recall than the other detectors, we aim to push its boundaries further. We observe that it has on average 227.6 usages examples (median = 105) for APIs whose misuses it identifies, but only 38.6 examples (median = 11) for those it misses. The moderate correlation (Pearson's $r = 0.52$) between the number of examples and detecting a misuse supports the hypothesis that the target projects lack usage examples for some APIs.

The 225 misuses in Experiment R are from 59 APIs. For five of these, we find no projects with respective usages on GitHub and for another thirteen, we find less than 50 projects. For the remaining 41 APIs, we find 50 or more projects. The cross-project sampling (Section IV-B) collects on average 239.3 usage examples per API (median = 172), compared to the average of 78.5 (median = 25) in the per-project setting.

The last row of Table III shows the precision and recall of MuDetectXP compared to those of the other detectors in Experiment P and Experiment R.

> *O4:* MuDetectXP reports 91 violations in the top-20 findings on the ten projects. Among these violations, we find 31 true positives, six of which were previously unknown. This results in precision of 33.0%, which outranks MuDetect by 11.1% and the other detectors almost fourfold.

> *O5:* MuDetectXP identifies 95 of the 225 misuses. This results in recall of 42.2%, which improves on MuDetect results more than twofold and on the other detectors' more than fourfold.

MuDetectXP identifies 65 misuses that MuDetect misses. For these misuses, MuDetect has on average only 94.6 usage examples (median = 16), while MuDetectXP has on average 258.4 examples (median = 216). This suggests that detectors should search for additional usage examples, if the target project itself contains too few. MuDetect, in turn, identifies 17 misuses that MuDetectXP misses. Ten of these are usages

of APIs declared in the respective target project. Interestingly, the problem is not a lack of examples, as MuDetectXP has on average 239.3 (median = 172). A possible explanation is that APIs are used differently in the declaring project than in client projects. This suggests that detectors should consider, but distinguish both sources of usage examples.

The results of Experiment XP show that mining patterns from other projects significantly improves both precision and recall (*P7 (Usage Examples)*). This is encouraging for API misuse detection researchers: given the completely automated pipelines provided by MuBenchPipe and MuBench, it should be straightforward for future work to integrate the latest techniques from finding reliable projects to mine [28] as well as evaluating quality of online examples [29]. This could even further improve on our results by easily retrieving more high-quality usage examples to train the detector.

F. Generalizability

Since we evaluate MuDetect on a dataset that we (in part) also used to design the detector, we run the risk of overfitting. To validate that this did not happen, we analyse by how much MuDetect improves over the other detectors separately on the original MuBench (MBO) and our dataset extension (MBE).

On average, MuDetect's precision increases 3.0x on MBO vs. 2.7x on MBE and MuDetectXP's precision increases 3.9x on MBO vs. 4.5x on MBE, showing that the precision improvement generalizes.

On average, MuDetect's recall increases 5.1x on MBO vs. 2.0x on MBE. This drop in recall improvement is due to MBE containing mostly smaller projects than MBO, where MuDetect's more precise analysis struggles with the small number of training examples. Training data is apparently crucial, because MuDetectXP's recall increases 5.5x on MBO vs. 6.6x on MBE, showing that the recall improvement generalizes, too.

G. Previously Unknown Misuses

In our experiments, MuDetect and MuDetectXP identified 27 previously unknown misuses. To validate these findings and as a contribution to the projects that served as our evaluation subjects, we manually created fixes for these misuses and submitted them as pull requests to the respective projects.

In this process, we excluded eight misuses, because the code containing them has been deleted from the respective project for reasons other than the misuse, and another three misuses, because the project does not accept pull requests. From the remaining 16 misuses, we created eight pull requests, grouping similar misuses into a single request.

To date, three of these pull requests were accepted: One fixes a bug in Google's Closure compiler, which caused it to crash on code with an invalid reference in a block comment. Such a reference led the compiler to access an empty `Iterator`, due to a missing check. Another fixes a bug in TestNG's XML export, which failed to close an XML tag along a specific execution path, leading to malformed data. The third fixes two bugs in Apache Lucene, which could lead search queries to crash due to missing checks on collections' length. This demonstrates that MuDetect finds relevant problems in mature projects.

H. Discussion

Our results show that MuDetect identifies relevant problems in mature software projects. It successfully adopts the strengths of existing detectors while mitigating many of their weaknesses, leading to 4x higher precision and recall. One of our industry partners showed interest to use MuDetect in code-quality audits. The most important design decisions to achieve this were (1) separating pattern mining and violation detection, which enables us to apply MuDetect in a cross-project setting, and (2) empirically investigating ranking strategies to push true positives to the top. Future work should investigate the performance and precision trade-offs of using inter-procedural static analysis to address remaining problems, such as *FP2* (Intra-procedural Analysis) and *FN1* (Self-Usages). Addressing other remaining problems, such as *FP1* (Uncommon Usages) and *FN2* (Redundant), likely requires different, e.g., probabilistic, models of API usage and mining algorithms.

VI. Threats to Validity

Overfitting. We designed MuDetect based on prior work's observations from experiments on MuBench [18]. We evaluated MuDetect (in part) on the same benchmark, which bears the danger of overfitting. To mitigate this threat, we extend the benchmark to more than twice its original size and validate that MuDetect's performance generalizes to this extended dataset.

Internal Validity. We did not fine-tune the other detectors, but used the best configurations reported in their respective publications. We reviewed the detectors' findings ourselves. The detector producing a finding was known, because we could not blind their distinct representations of API usages and violations. We evaluated only MuDetectXP in the cross-project setting, because the other detectors cannot use separate datasets for mining and detection. Modifying them ourselves to support this might hamper with their capabilities. We published the list of example projects we used [22] and encourage others to assess their approach in this setting. Providing MuDetectXP with only example usages for the APIs with known misuses in MuBench potentially biases the results with respect to precision, because it reduces the overall number of patterns and, consequently, might reduce the number of reported violations.

External Validity. The dataset of API misuses in our evaluation might not be representative. We mitigated this by using MuBench, a public and state-of-the-art benchmark. The API misuses it contains cover the capabilities of all detectors in our evaluation. We further extend the benchmark by findings from a large-scale study [19].

VII. Related Work

Helping developers use APIs has received much attention. Approaches include improving documentation (e.g., [30], [31]) and assisting developers with recommendations while writing code (e.g., [32], [33]). Another direction is API-misuse detection, which can be further classified into static and dynamic approaches. Dynamic approaches execute programs to detect deviations from normal behavior (e.g., [34], [35]). Our focus is on static misuse detectors, so we briefly discuss existing ones.

Amann *et al.* [18] present a detailed survey and comparison of detectors and their capabilities.

The closest work to MuDetect is GrouMiner [12], which uses a graph-based representation (GROUMs) for API usages. GrouMiner's relatively high recall [18] led us to also use a graph representation. Both AUGs and GROUMs are directed graphs that capture calls, field accesses, and control/data dependence. However, GROUMs are simple graphs that encode actions and loop/branching statements in nodes and use unlabelled edges to uniformly represent data and control dependence. AUGs are multigraphs that capture actions and data entities in nodes and distinguish different kinds of control/data dependence (including exceptional and synchronized flow) in labelled edges. This precisely differentiates usages in our mining and detection algorithms and improves scalability.

In addition to the detectors we compare to, there are detectors for various languages. For C, detectors include: PR-Miner [8] uses frequent-itemset mining to detect missing method calls. Colibri/ML [9] re-implements PR-Miner using Formal Concept Analysis [23]. RGJ07 [11] uses frequent-itemset mining to detect missing call-argument conditions. Chronicler [36] mines frequent call-precedence relations to detect call-order violations. AX09 [13] uses push-down model checking to detect missing error handling.

Car-Miner [14] is a detector for C++ and Java, specialized in detecting wrong error handling. Alattin [15] is a detector for Java that detects missing `null` checks, missing value or state conditions not involving literals, and missing calls required in checks. DroidAssist [17] is a detector for Dalvik Bytecode. It uses a Hidden Markov Model to compute the likelihood of call sequences to detect missing, misplaced, and redundant method calls; no evaluation was presented in the paper.

VIII. Conclusion

In this paper, we investigate whether the performance of API-misuse detectors can be improved. We design MuDetect to build on the strengths, and address many of the problems, of existing detectors. We systematically design a ranking strategy that effectively ranks true positives among MuDetect's top findings. We compare the performance of MuDetect to four state-of-the-art detectors. Our evaluation shows that MuDetect clearly outranks these detectors, with recall upper bound of 72.5%, recall of 20.9%, and precision of 21.9% in the typical per-project setting. In a cross-project setting, MuDetect's recall reaches 42.2% and its precision 33.0%. We also analyze the remaining false negatives and false positives to help researchers identify further improvement opportunities.

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[17] T. T. Nguyen, H. V. Pham, P. M. Vu, and T. T. Nguyen, "Recommending API usages for mobile apps with Hidden Markov Model," in *Proceedings of the 30th ACM/IEEE International Conference on Automated Software*

Engineering, ser. ASE '15. IEEE Computer Society Press, 2015, pp. 795–800.

[18] S. Amann, H. A. Nguyen, S. Nadi, T. N. Nguyen, and M. Mezini, "A systematic evaluation of static API-misuse detectors," *IEEE Transactions on Software Engineering*, 2018.

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

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

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

[22] "Artifact Page," 2019. [Online]. Available: http://www.st.informatik.tu-darmstadt.de/artifacts/mudetect/

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

[24] T. Ramraj and R. Prabhakar, "Frequent subgraph mining algorithms – a survey," *Procedia Computer Science*, vol. 47, pp. 197–204, 2015.

[25] H. A. Nguyen, T. T. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen, "Accurate and efficient structural characteristic feature extraction for clone detection," in *Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering*, ser. FASE '09. Springer-Verlag, 2009, pp. 440–455.

[26] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen, "Boa: A language and infrastructure for analyzing ultra-large-scale software repositories," in *Proceedings of the 35th International Conference on Software Engineering*, ser. ICSE '13. IEEE Computer Society Press, 2013, pp. 422–431.

[27] T. T. Nguyen, H. V. Pham, P. M. Vu, and T. T. Nguyen, "Learning API usages from bytecode : A statistical approach," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. ACM Press, 2016.

[28] N. Munaiah, S. Kroh, C. Cabrey, and M. Nagappan, "Curating GitHub for engineered software projects," *Empirical Software Engineering*, vol. 22, no. 6, pp. 3219–3253, 2017.

[29] S. Radevski, H. Hata, and K. Matsumoto, "Towards building API usage example metrics," in *Proceedings of the 23rd International Conference on Software Analysis, Evolution, and Reengineering*, ser. SANER '16. IEEE Computer Society Press, 2016, pp. 619–623.

[30] C. Treude and M. P. Robillard, "Augmenting API documentation with insights from Stack Overflow," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. ACM Press, 2016, pp. 392–403.

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

[32] M. Bruch, M. Monperrus, and M. Mezini, "Learning from examples to improve code completion systems," in *Proceedings of the 7th ACM Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC/FSE '09. ACM Press, 2009, pp. 213–222.

[33] R. Holmes and G. C. Murphy, "Using structural context to recommend source code examples," in *Proceedings of the 27th International Conference on Software Engineering*. IEEE Computer Society Press, 2005, pp. 117–125.

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

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

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