A Massive Study on API Misuses in the Wild and Its Implications

Anonymous Author(s)

ABSTRACT

API misuses are prevalent and extremely harmful. Despite various static and dynamic techniques have been proposed for API-misuse detection, it is not even clear how different types of API misuses distribute and whether existing techniques have covered all major types of API misuses. Therefore, in this paper, we conduct the first massive empirical study on API misuses based on 958,368 historical bug-fixing commits on GitHub (from 2011 to 2018). By leveraging a state-of-the-art fine-grained AST differencing tool, GumTree, we obtain more than six million bug-fixing edit operations, 50.6% of which are API-related misuses. We further systematically classify the API misuses into nine different categories according to the code-change and context information. Our study reveals various practical guidelines regarding the importance of different types of API misuses: among the API misuses covered by prior techniques, API call (including missing and redundant ones) is the most frequent (38.23% of all API misuses) while synchronization is the least frequent one (0.05% of all API misuses). Surprisingly, replaced API misuses, not covered by any of prior techniques, account for 38.19%. Furthermore, the massive study also reveals various frequent API-misuse patterns (e.g., `java.io.File.mkdir()=>java.io.File.mkdirs()`) from the big-data perspective, which can easily be used to design our big-data-driven API-misuse detection technique (BiD^3). Our additional experimental results show that (1) seven API misuses that no state-of-the-art techniques can detect in the widely used MuBench exist in our mined patterns, (2) for the replaced API misuses that no existing techniques can detect, BiD^3 detects 360 previously unknown API misuses in Apache projects (149 of them are reported to developers and 57 of them have already been confirmed thus far). The promising results indicate the importance of studying historical API misuses and the promising future of big-data-driven API-misuse detection.

ACM Reference Format:
Anonymous Author(s). 2020. A Massive Study on API Misuses in the Wild and Its Implications. In *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2020).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Over the past decades, software systems have been widely adopted in almost all aspects of human lives, and are making our lives more and more convenient. However, software systems also inevitably suffer from bugs or faults, which can incur great loss of properties and even lives. During modern software development, developers always reuse Application Programming Interfaces (APIs) provided by third-party libraries and frameworks rather than implementing from scratch to improve work efficiency and code quality. As a result, API related bugs spread widely due to API misuses, reducing software performance or causing software crashes [13, 32]. In Java, for example, the correct way to create new thread is to call API `Thread.start()` while `Thread.run()` is often misused since it is executed on current thread without creating any new thread. Therefore, analyzing the behaviors of API misuses is essential and can provide practical guidance for software development, especially the detection of API misuses. Such misuse behaviors can be obtained from historical bug fixes because developers tend to misuse APIs in the buggy files while correcting such misuses in the fixed files.

A large number of historical bug fixes have been collected in version control systems [3, 10] and/or issue tracking systems [8]. Many researchers have conducted various studies to analyze bug fixes and have a number of findings. For example, Martinez and Monperrus [29] found that most of bug fixes are related to more than one source file and statement-level code changes (e.g., inserting or deleting a statement) are most prevalent by analyzing 89,993 historical bug fixes. Even though these studies have studied historical bug fixes, they only analyze a small amount of dataset and the findings from the studies are limited. In this paper, we conduct a massive study related to API misuses by analyzing much more comprehensive bug-fixing dataset than any prior work.

There exist many different techniques to study API misuses and how to effectively detect API misuses. Typically, they tend to mine API usage patterns which represent that certain API usages occur frequently and violations of such patterns are regarded as API misuses. Various techniques have been proposed to detect different types of API misuses using different algorithms [11, 26, 43, 44, 48]. Even though these techniques can effectively detect many types of misuses, one recent study on state-of-the-art API-misuse detectors [13] shows that API misuses are still prevalent. This study also shows that existing API-misuse detectors suffer from extremely low precision and recall on the recent widely used API-misuse dataset MuBench [12]. MuBench includes only 89 API misuses from 33 real-world projects, which not only makes it hard to analyze the distribution of different API misuses but also incurs dataset overfitting issue for evaluating existing API-misuse detection techniques.

Due to the limited amount of dataset in previous studies of bug-fixing and API misuses, in this paper, we conduct a more systematic and extensive empirical study on the distribution of various real-world API misuses in the wild by analyzing a large-scale bug-fixing dataset. We start with mining the historical repositories from a GitHub Archive [4] which records the public GitHub timeline dataset. We extract all bug-fixing commits of Java projects from 2011 to 2018 according to certain search criteria. This results in 958,368 pairs of bug-fixing commits. We download pairs of fixed files as well as corresponding buggy files from all these commits.

ISSTA 2020, 18-22 July, 2020, Los Angeles, US

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Extracting the differences between fixed and buggy files can help us understand the API misuses. To achieve this goal, we use a fine-grained source code differencing tool [16] to compute edit operations on AST (Abstract Syntax Tree) of Java code between the buggy and fixed files, such as updating an API to another, inserting or deleting an `if` condition. To better understand API misuses, following existing studies [13], we classify API misuses into different categories based on extracted edit operations and statistically analyze the distributions for each category. From the statistical results, we observe various practical guidelines regarding the importance of different categories of API misuses, e.g., among the API misuses covered by prior techniques, API call (including missing and redundant ones) are the most frequent (38.23% of all API misuses) while synchronization is the least frequent (0.05% of all API misuses). We also find that one category of misuse related with replacement of API components (i.e., arguments, receiver or name) has never been systematically explored by any of the prior techniques, counting for 38.19%. According to the statistical results, we extract a large set of frequent API-misuse patterns for each category. Based on these patterns, we carry out another study on detecting API misuses in two aspects. On one hand, following previous studies [13, 49], we analyze how many API-misuse patterns in MuBench can be found in the historical bug fixes and compare the results with existing state-of-the-art techniques. The results show that 12 out of 32 unique repetitive patterns in MuBench can be found in the historical commits, corresponding to 22/53 API misuse instances in the benchmark. Furthermore, 7/12 unique API-misuse patterns can not be detected by any of the existing techniques, demonstrating the importance and possibility to provide guidance for building more effective bug detection tools via analyzing massive historical bug fixes. As a consequence, we implement a big-data-driven API-misuse detection technique, BiD^3, to apply 10 replaced API misuses extracted into detecting unknown API misuses in Apache projects. The study shows that the replaced API misuses extracted from historical records still exist in latest projects. More specifically, we scanned 688 Apache projects and detected 360 API misuses. To date, we have reported 149 misuses to developers and 57 have already been confirmed and fixed. Such promising initial results indicate that the misuse patterns can be complementary to existing static API-misuse detection tools and potentially can improve the practicability of bug detection approaches.

In summary, this paper makes the following contributions:

- **Study.** A massive study on GitHub historical bug-fixing data to analyze the categorization and distribution of API misuses in the wild.
- **Dataset.** A publicly available dataset including 3,197,593 API misuses of various types that can help evaluate and boost future API-misuse detection.
- **Guidelines.** Various practical guidelines regarding the importance of different types of API misuses, e.g., among the API misuses covered by prior techniques, API call is the most frequent (38.23% of all API misuses) while synchronization is the least frequent (0.05% of all API misuses), and replaced API misuses, not covered by any of the prior techniques, account for 38.19%.
- **Patterns.** A large set of mined API-misuse patterns based on the big data of API misuses, which can help detect unknown API misuses in the wild.
- **Technique.** An initial big-data-driven API-misuse detection technique, BiD^3, is implemented to detect various previously unknown misuses on Apache projects (57 of them are already accepted by developers), indicating a promising future for big-data-driven API-misuse detection.

2 BACKGROUND AND RELATED WORK

In this section, we first introduce previous studies on historical bug fixes and then discuss existing techniques working on API-misuse detection.

2.1 Analysis on Historical Bug Fixes

During modern software development, a large number of historical bug fixes get accumulated in version control systems [3, 10] and/or issue tracking systems [8]. Understanding and analyzing the historical bug fixes can potentially provide practical guidelines for manual or automated bug detection, localization, and repair. Therefore, various studies have been conducted on historical bug fixes. Martinez and Monperrus [29] studied 89,993 historical bug fixes from various open-source projects and built a classification model to understand the search space of automatic program repair under different models. In the study, they found that (1) most of bug fixes are related to more than one source file, and (2) statement-level code changes (e.g., inserting or deleting a statement) are the most prevalent bug-fixing pattern, providing useful guidance for automatic program repair techniques. Zhong and Su [52] built a bug-fixing extraction tool named BugStat and analyzed more than 9,000 real bug fixes from six popular open-source Java projects. They found that most bug fixes only update existing source code files and do not add or delete source files, which is consistent with the findings by Martinez and Monperrus. Additionally, they found that most bug fixes are related to `if` conditions, which is also confirmed by Soto et al. [42]. Pan et al. [38] also analyzed the distribution of different bug-fixing patterns from seven open-source projects. They found that updating method call parameters and `if` conditions are the most common bug fixes.

Besides, there were also studies on domain specific bug patterns or bug pattern distribution cross different projects. Wan et al. [46] used the card-sorting approach to analyze the characteristics of bugs in Blockchain systems. Meng et al. [30] conducted an empirical study on StackOverflow posts related to code security, which revealed the huge gap between security theory and coding practices and informed effective secure coding assistance. Similarly, Hanam et al. [18] studied the bug patterns in JavaScript projects, and found that the same bug-fixing patterns exist among different JavaScript projects. Additionally, this finding was further confirmed on Java projects by Yue et al. [51] and Nguyen et al. [33]. Similarly, Ray et al. [41] found that although source code is highly repetitive and predictable (like natural languages), the buggy code tend to be unnatural. After comparing with different statistic models, they found that "entropy" is a relatively good model to measure the similarity between code fragments, which can be used in search-based bug-fixing approaches.

Although various existing studies have already conducted on general software bugs, the API misuses have not been systematically explored yet. Therefore, in this paper we aim to perform a massive study on the categorization and distribution of API misuses in the wild, which complements existing research. Furthermore, to the best of our knowledge, our study involves 958,368 historical bug-fixing commits from GitHub open-source projects, and represents the largest study on historical bug fixes to date.

2.2 Studies on API-misuse Detection

API usage is often subject to certain constraints [13]. For example, a resource must be released or closed after it is used. Violations of such usage patterns are regarded as API misuses. Therefore, a large number of techniques have been proposed to detect API misuses automatically over the past decades.

Most techniques utilize data mining information to detect API misuses. Livshits et al. [28] introduced DynaMiner, aiming to mine software revision histories to detect misuses violating method pairs or certain mined state machines. Similarly, Li and Zhou [23] proposed PR-Miner to extract API implicit programming rules by leveraging frequent itemset mining on source code, while Acharya and Xie [11] proposed to mine specifications from static program traces. Although different data sources are utilized by such techniques, they are sharing the same assumption that the more frequent a pattern is, the higher possibility a pattern holds to be correct. Other similar techniques also include DMMC [31, 32], GrouMiner [35], COLIBRI/ML [26], etc. In addition, both Wasylkowski et al.[48] and Nguyen et al. [34] proposed to employ graph theories for mining programming artifacts.

Researchers have also proposed data-mining-based techniques to detect other specific types of API misuses. Williams et al. [50] and Hovemeyer et al. [20] targeted missing NULL pointer checks, while Thummalapenta and Xie focused on exception-handling related misuses [44] and neglected-condition misuses [43]. More recently, Liang et al. [25] aimed to detect missing NULL pointer and resource-leaking misuses (e.g., missing API invocations to close resource accesses) via analyzing existing bug fixes of the same projects.

Additionally, besides data mining, researchers also employed program analysis and machine learning for API-misuse detection. Ramanathan et al. proposed CHRONICLER [39] and RGJ07 [40], which utilize path-sensitive control-flow or data-flow analysis to infer function precedence protocols or predicates. Wasylkowski and Zeller [47] proposed TIKANGA to combine static analysis with model checking for mining Computation Tree Logic (CTL) formulas, while Nguyen et al. [36] leveraged Hidden Markov Model to check anomalies of call sequence. Most recently, Wen et al. [49] applied mutation analysis to discover API misuse patterns to improve the state-of-the-art.

Recently, Amann et al. [12] presented MuBench, a dataset of 89 API misuses from 33 real-world projects and a survey about API misuses in practice. Later, Amann et al. [13] further empirically evaluated various API-misuse detectors on the MuBench dataset, but only considered missing and redundant API invocations, revealing the low recall and precision of existing API-misuse detectors.

Although various techniques have been proposed for detecting different types of API misuses, it is not even clear how different types of API misuses distribute among all API misuses or projects and whether the existing techniques have covered all major types of API misuses. Furthermore, the recent widely used API-misuse dataset MuBench [12] includes only limited number of API misuses, and is insufficient for evaluating API-misuse detection techniques. Therefore, in this work, we aim to perform a systematic and extensive empirical study to characterize the distribution of various types of API misuses in the wild, and construct a massive dataset for API-misuse detection.

3 EMPIRICAL STUDY

In this section, we introduce how we construct our dataset and conduct our study. We first introduce the collection of dataset used in our study (Section 3.1), and then introduce the categorization of API misuses in our study (Section 3.2). Finally, we discuss how we applied source-code differencing to infer API misuses from bug fixes (Section 3.3).

3.1 Data Collection

3.1.1 Dataset for Mining API-related Misuse Patterns. We aim to mine API-misuse patterns from all bug-fixing commits of all Java projects on GitHub [6]. To collect our dataset for API-misuse patterns, we first download all public GitHub events for all program languages from GitHub Archive [4][1] between 2011 and 2018. We then focus on Java projects and exclude all test cases since they are not functional parts and cannot reflect API usages. Next, we identify a commit as bug-fixing if its message contains the keywords ("fix" or "solve" or "repair") and ("bug" or "failure" or "issue" or "error" or "fault" or "defect" or "flaw" or "glitch") following prior work [45]. Since the commit message may not identify bug-fixing commits accurately, we extract 100 commit samples randomly and two authors independently analyze them to check whether they are actual bug fixes. Finally, we conclude that 94% of the identified bug-fixing commits are real bug fixes. Then we use GitHub API [5] to retrieve the previous commit of each collected bug-fixing commit to construct a pair of buggy and fixed commits. For each pair of commits, we download the source files to create a pair of buggy files and fixed files. We filter out none-Java files because this work only focuses on Java API misuses. We also discard the files existing in only one commit since they were deleted or newly added in the bug-fixing commit, which cannot reflect API misuses. Also, following previous work [45], we discard commits with more than five Java files, since such commits may include many benign changes. After the filtering processes, we get 958,368 bug-fixing commits including pairs of buggy and fixed Java files for further API-misuse pattern mining. Figure 1 shows the commit numbers of each year from 2011-2018.

3.2 Categorization of API Misuses

Bug-fixing commits can involve different modifications by developers to fix the bugs. If developers add a guard condition to check if one variable is NULL, the bug can be classified as a misuse of *missing* guard condition. On the contrary, if developers fix a bug by deleting an unnecessary API call, the bug indicates a *redundant* API misuse. Using the two *violation types* (missing and redundant),

[1] GitHub Archive is a web service that provides all public GitHub revision histories.

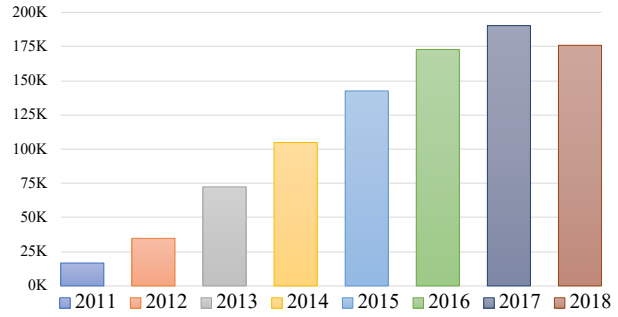

Figure 1: Commit numbers from 2011-2018.

recent study [13] has investigated API misuses involving different underlying elements (e.g., API call, iteration, condition and exception handling) on the MuBench benchmark. In this paper, we employ most of the definitions in the above study. Please note that we do not consider the "iteration" category in our classification since it is hard to automatically identify whether the code change is related to iteration or not. For example, as presented in the previous study, API call `wait()` on an object should always occur in a loop, which is stated in the Java document. It is easy to identify this API call by manual inspection [13], while it will be infeasible in our study with a massive dataset since it is hard to automatically analyze such specifications of each API usage from informal documents which are usually presented via natural language. Besides these existing categories, API misuses of invoking the *wrong* APIs have not been categorized before. To fix such bugs, the developers tend to replace the wrong API invocation with another one (with different name, argument, or receiver objects). In this work, we also study such *replaced* APIs systematically. In summary, we categorize all API misuses into the following categories in this study:

Condition. This category includes missing and redundant guard conditions for certain API invocations. Following the previous study [13], we further categorize it into the following three sub-categories:

- *NULL checks.* This sub-category indicates the removal or addition of conditions with NULL checks for variables that is returned by a prior API call or is used as the receiver or an argument of a following API call, e.g., `o.API(); => if(o!= null){ o.API();}`.
- *Return value.* This sub-category indicates the removed or newly introduced `if` condition checks the return value of some APIs, e.g., `o = API(); a = list.get(o); => o = API(); if (o < 0){o = 0;} a = list.get(o);`.
- *Object state.* This sub-category indicates the removed or newly introduced `if` condition is related to some variable that will be used in some API calls immediately, and thus it should meet certain constraints, e.g., `a = list.get(i); => if(i > 0){a = list.get(i);}`.

Please note that these three sub-categories of `Condition` are not orthogonal to each other as one `if` condition may belong to multiple categories. We will introduce this in detail in Section 3.3.

Exception. This category includes missing and redundant exception handling, following the definitions by the prior work [13]. More specifically, we further divide this category into two types

```
// replaced arguments
--- row=Math.abs(rand.nextInt(seed)%data.length-1);
+++ row=Math.abs(rand.nextInt()%(data.length-1));
// replaced name
--- nVal=tmp1.substring(0,tmp1.indexOf("\""));
+++ nVal=tmp1.substring(1,tmp1.lastIndexOf("\""));
// replaced name and arguments
--- Statement stmt = con.createStatement();
+++ PreparedStatement stmt = con.prepareStatement(sql);
// replaced receiver
--- return this_path.equals(that_path);
+++ return Objects.equals(this_path, that_path);
```

Figure 2: Examples of replaced bugs.

of fine-grained code changes, i.e., inserting or deleting `Try` or `Catch` blocks. The reason is that in the studied commits, we found that some fixes are related to a complete `try-catch` statement, but some others may only involve `catch` blocks. Therefore, we analyze them separately. Especially, we regard a `try` or `catch` as API-related *iff* there exist API calls in the corresponding code block; otherwise, we consider it as API non-relevant. Besides, like the `Condition` category introduced before, a code change may involve both `try` and `catch` blocks. In such cases, we record these two categories respectively.

- *Try.* This category subjects to addition or deletion of `try` blocks, in which some API invocations reside.
- *Catch.* This category indicates that there is an API invocation within the added or removed `catch` blocks.

Synchronization. This category includes missing and redundant synchronizations in multi-threaded environments, following the previous study by Amann et al. [13]. The difference is that we classify this type of code change as an independent one rather than one of *Condition* changes (introduced before).

API call. Previous studies have focused on missing [23, 31] and redundant [36] API call misuses. However, the more fine-grained API changes (such as changing only the arguments, names or receiver objects of API invocations) are not considered by any existing study. In this paper, besides those missing and redundant API changes, we further investigate the distributions of replaced API misuses, which includes the following four categories in detail. For the missing and redundant API changes, previous study [13] has already introduced (a.k.a. Method Call). To make the article self-contained, we redundantly explain them here briefly.

- *Missing & Redundant API Call. Missing API Call* denotes that an API is not called at a certain place, where the API usage constraint requires the API as a must. For example, after opening a file and writing data, the API of `File.close()` should be called. Otherwise, errors would be incurred. This kind of code changes is usually related to those pair-wised APIs that have usage dependency. Similarly, *Redundant API* represents that some API is redundantly used at an improper place. For example, we cannot call the API of `List.remove()` to delete elements in a list that is being iterated over. Otherwise, exceptions would be raised. This kind of code changes is usually caused when the function of the API has side-effect, whose execution may conflict with the following functionality.
- *Replaced arguments.* This category indicates that developers may pass incorrect arguments or arguments with wrong orders when invoking an API. This type of code changes

usually appears in classes with multiple methods with similar functionalities for polymorphism, such as the example shown in Figure 2, where the desired API is `nextInt()` without argument. On the contrary, a wrong API `nextInt(int)` was used with an argument `seed`, which will constrain the upper bound of the generated random value. Please note that we consider replaced API misuses *iff* the types of arguments do not match before and after the change (type order is considered). For example, the code change replacing `10` in `nextInt(10)` with `100` is not regarded as a replaced API misuse since the argument type is not changed and thus the API is not changed.

- *Replaced name.* This category implies that developers call a wrong API with exactly the same arguments but different API names within the same class. This kind of misuse is usually caused by the similarity or confusion of different API names. Figure 2 shows one example, in which the API call of `indexOf` is replaced with `lastIndexOf` and the argument stays unchanged.
- *Replaced name and arguments.* This category implies that the whole API including its name and arguments are called incorrectly, which can be introduced when developers are not familiar with the class under use and mistakenly pick an improper API in the same class. For example, when a database is frequently accessed in a loop with the same query, `prepareStatement(String)` rather than `createStatement()` should be used to avoid high overhead as it will be pre-compiled by the database management system (*ref.* Figure 2).
- *Replaced receiver.* The above three replaced API misuses are all related to API misuses within the same class. However, sometimes developers even misuse APIs belonging to different classes. We classify this kind of replacements as *replaced receiver*. As introduced before, we only consider the type change of receiver. For example, the code change in Figure 2 shows that the developer fixed the bug by using the `equals` API in the `Object` class for `String` comparison rather than comparing them directly. The reason is that `this_path` is not guaranteed to be properly set, and it may cause `NullPointerException` and crash the program.

3.3 Edit Operation Extraction

To study different categories of misuses defined in Section 3.2, we first need to collect code changes from historical bug-fixing commits. In our study, we apply a state-of-the-art AST (Abstract Syntax Tree) differencing tool GumTree [16], which can extract fine-grained AST operations and has been widely employed by previous studies [21, 22, 27]. According to the classification definitions, we consider three types of operations defined in it, i.e., *update*, *delete* and *insert*. The three operations respectively correspond to the three types of API misuses: replaced, redundant and missing APIs defined in this paper. Especially, we do not consider *move* operation and we will explain this in detail later. Then, according to the content of changed code and the type of operation, we try to map AST operations to the corresponding misuse category. Please note that a pair of bug-fixing commits may include multiple operations if the changed code involves more than one AST node.

```
static List<String> writeFiles(State state,...){
    ...
--- fPath=fPath.replaceAll("/",File.separator);
+++ fPath=fPath.replace("/",File.separator);
    ...
    if (!isLoaded) {
        Log.e("WavefrontLoader","Error loading");
---     System.exit(1);
    }
}
```

Figure 3: Examples of *update* and *delete* operations.

```
public void setAnimationStyle(int animRes) {
     Window window = dialog.getWindow();
+++  if (window != null) {
         window.setWindowAnimations(animRes);
+++  }
}
```

Figure 4: Examples of *insert* and *move* operations.

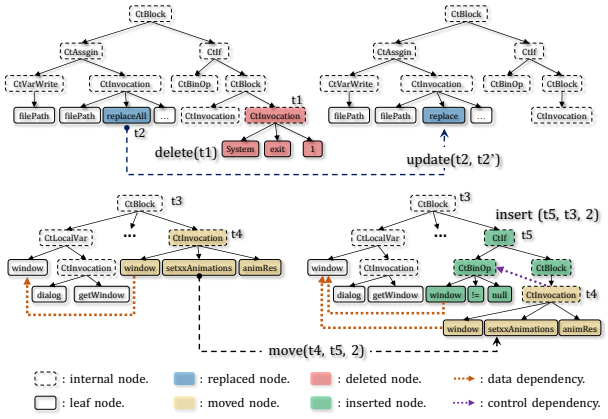

Figure 5: GumTree operations of examples in Figures 3-4.

However, it is not straightforward to map GumTree operations to our API misuse categories since those operations do not comprise context information related to the code changes, such as data and control dependency. Therefore, we further implement a lightweight intra-procedural data- and control-dependency analysis. More concretely, to identify whether an inserted `if` condition is checking the return value of some API, we need to analyze the Use-Define chain [17, 19] for each variable used in the condition: the operation can map to the type of "Return value" condition only if some variables are the return values of some early API invocations. Similarly, to identify the "Object state" condition, both data and control dependencies will be utilized. As a result, we combine the result of GumTree and our lightweight dependency analysis to determine the API-misuse categories of bug-fixing operations.

Before introducing the detailed classification process, we first introduce some preliminary concepts and notations:

Definition 3.1. An abstract syntax tree (AST) is a partial order tree whose root node can be represented as a tuple $\langle l, v, p, i, C \rangle$, where

- l: denotes the label of the root node of the subtree. (e.g., StringLiteral.)
- v: saves the value if it is a leaf node, otherwise is $\perp$. (e.g., 4.)

- p: represents its parent node in a super tree if exists, otherwise is $\perp$.
- i: is the index of the root node in a super tree p, it is undefined if $p = \perp$.
- C: contains a sequence of immediate child node in the subtree, it will be $\emptyset$ for leaf node.

Finally, based on the description of operations in GumTree [16], it will be straightforward to give the operation definitions under the AST definition.

Definition 3.2. A GumTree operation is one of the following AST node changes:

- $update(t, t')$: replace the subtree rooted t with a subtree rooted t';
- $delete(t)$: deletes subtree rooted node t.
- $insert(t, t', i)$: adds a new node t as the i^{th} child of node t' if t' is not $\perp$. Otherwise, t is the new root node and the previous root node will be the only child of t.
- $move(t, t', i)$: moves subtree rooted node t to be the i^{th} child of node t'.

As introduced at the beginning of this section, we only adopt the first three kinds of operations for API misuses. To do so, we perform a preprocessing for $move(t, t', i)$ operation, which involves three situations: (1) If t' is a newly inserted node, we simply discard the *move* operation because the node t is unchanged but a new internal node t' is inserted into the original AST. (2) If t' is an ancestor node of t in the original AST, we discard the *move* operation as well because the change should be related to a deletion of some internal node that is the parent of t and child of t'. (3) Otherwise, if t' is the same as $parent(t)$ but i is different from $index(t)$, i.e., changing the index of t with its sibling nodes, we consider this *move* operation is not API-misuse because it is hard to identify whether it is a logical misuse. As a result, for API misuses, we only consider the first three kinds of operations.

Next, we will use the operations shown in Figure 5 as examples to demonstrate the mapping process in detail. According to the explanation for each category in Section 3.2, the operation $delete(t1)$ will be classified as *redundant* API Call because the call to API `System.exit(int)` is deleted to fix the bug, while the operation $update(t2, t2')$ will be classified as *Replaced name* of API Call. Please note that if there exist another operation on the arguments of the same method (i.e., replaceAll()), such as inserting an additional argument or change the argument type, we will combine these two operations together and classify them as one *Replaced name and arguments* (e.g., updating `createStatement()` to `prepareStatement(String)` in Figure 2). For the operation $move(t4, t5, 2)$, it belongs to the first situation introduced above, where a new node (i.e., $t5$) is inserted and the moved node becomes one of its child node. Therefore, we discard the move operation and only analyze the insertion $insert(t5, t3, 2)$. From the figure we can see that a `null` check condition to variable `window` is inserted, which is the returned value of API `getWindow()`. As a consequence, it will be classified as both missing NULL checks and Return value. Additionally, variable `window` is further used by API `setWindowAnimations()` that has control-dependency on the condition related to `window`. As a result, it will be classified as Object state as well. Consequently, one operation may classified into multiple categories in **Condition**, but other categories will be mutually exclusive.

4 EMPIRICAL RESULT ANALYSIS

According to the previous sections, we collect a massive API-related misuses in real-world projects. In this section, we conduct various empirical studies on them and discuss the results.

4.1 How Often Do Developers Introduce and Fix API Misuses?

In this research question, we first compare the number of raw edit operations between API-related and none API-related misuses. From the result, we can find that more than half (50.6%) of or more than three million studied operations involve API misuses. This finding shows that developers tend to introduce API misuses frequently in modern software development. One potential reason is that developers are using more and more 3rd-party libraries to save development efforts and improve code quality. To our knowledge, this is the first study quantitatively demonstrating the importance of API-misuse detection in modern software systems.

Table 1: Distribution of different API misuses

Category		Missing	Redundant
API call		880,584 (27.54%)	341,969 (10.69%)
Synchronization		1,304 (0.04%)	332 (0.01%)
Condition	NULL checks	38,378 (1.20%)	7,265 (0.23%)
	Return value	74,933 (2.34%)	16,798 (0.53%)
	Object state	107,255 (3.35%)	22,684 (0.71%)
	Total	220,566 (6.90%)	46,747 (1.46%)
Exception	Try	25,001 (0.78%)	6,314 (0.20%)
	Catch	29,586 (0.93%)	8469 (0.26%)
	Total	54,587 (1.71%)	14,783(0.46%)
Replaced API	Rep Receiver	638,220 (19.96%)	
	Rep Name	195,099 (6.10%)	
	Rep Args	258,807 (8.09%)	
	Rep Name&Args	129,020 (4.03%)	
	Total	1,221,146 (38.19%)	

Table 1 presents the distributions of different categories of API misuses based on the misuse category presented in Section 3.2. Please note that we separate replaced APIs with missing and redundant APIs since it is one of the primary research topics in our paper. In the table, the first column represents different misuse categories according to Section 3.2. The last two columns present the raw number (ratios to all API misuses presented in brackets) of missing and redundant misuses, respectively. Especially, we directly present the number of replaced API misuses. From this table, we have the following findings.

First, API misuses without other types of contextual information are more prevalent. The ratios of all missing, redundant and replaced API misuses which do not refer to other code elements such as condition, synchronization and exception are more than 70%. For example, missing and redundant API misuses account for 27.54% and 10.69%, respectively, i.e., almost 40% of all API misuses. Also, the ratio of replaced API misuses is 38.19%, larger than that of any other types of API misuses too. The possible reason of such high portions of replaced misuse is that many APIs share similar appearance or functionalities, making developers call incorrect APIs very frequently. For example, we have mined a misuse pattern `Date.getTime()=>System.currentTimeMillis()` from our dataset.

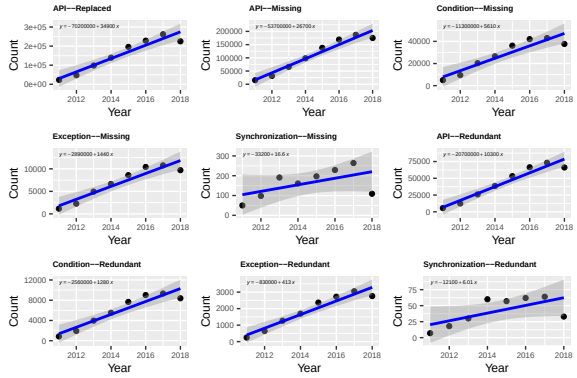

Figure 6: Counts of different categories in different years

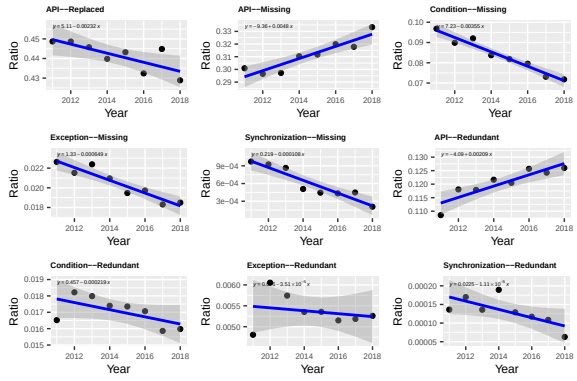

Figure 7: Ratios of different categories in different years

This pattern illustrates that `Date.getTime()` can be replaced with `System.currentTimeMillis()` in some cases. One reason is that a `Date` object is required for `getTime()`, and `System.currentTimeMillis()` can be called directly. Another reason is that `System.currentTimeMillis()` is inside the default constructor of Class `Date`. Therefore, if a `Date` object is not used in other places, it is better to directly use `System.currentTimeMillis()` to speed up the underlying system. Currently, 10 replaced misuses detected in this work have been accepted by developers. This finding demonstrates the importance of replaced API misuses for the first time, and more research efforts need to be dedicated to this overlooked API-misuse category.

Second, developers tend to miss API invocations or handling (such as condition checks, synchronization, and exception handing) rather than having redundant ones. We can observe that missing API call, synchronization, and condition are all more than the redundant ones. For example, the ratio of missing API calls is 27.54%, twice higher than redundant API calls. Also, missing condition has a ratio of 6.9%, which is higher than that of redundant condition (1.46%). This finding implies that developers often forget to invoke certain APIs or add some necessary handling for the corresponding APIs, such as NULL checks. This can also provide practical guidance for both researchers and developers.

4.2 How Do API Misuses Distribute Over Time?

In the first RQ, we have a big picture about different API-misuse categories in all historical GitHub bugs. In this RQ, we are interested in how these different API-misuse categories evolve over time to help make potential prediction in future. To this end, we analyze the trends of all general misuse categories API call, synchronization, condition and exception in terms of raw counts and ratios to all API misuses from 2011 to 2018. Figure 6 shows the raw-count trend and Figure 7 shows the trend for the proportion of each category to all API misuses within the same year. The two figures share similar format. In each figure, the *x-axis* represents the studied years from 2011 to 2018. The *y-axis* represents the corresponding counts or ratios. We also draw regression lines to show the trends across different years. From the two figures we have following findings.

First, the raw-count trends for all API misuses keep increasing year by year. For example, in terms of raw Gumtree edit actions, the number of replaced API misuses increases from around 23,000 to 225,000 between 2011 and 2018. The count of missing exception also increases from 1,163 to 9,709. This result is not surprising since the total number of GitHub projects keeps increasing along with the development of information technology, introducing more and more misuses. However, we can still find some differences in terms of increasing rate. The increasing rate of synchronization is much smaller than those of other categories. For example, the increase rate of missing condition is 600% (from 4974 to 37691) between 2011 and 2018. However, for missing synchronization, the increasing rate is only 100%. This finding shows that developers make fewer mistakes to synchronization API misuses. There are several potential reasons: (1) developers using advanced APIs involving synchronization are usually experienced developers, (2) synchronization misuses are rare in practice (e.g., we only see hundreds of such instances).

Second, we can find that only the proportions of missing and redundant API call increase (from Figure 7). For example, the ratio of missing API call increases from 30% to 33.3%. This observation indicates that it becomes harder to use API sequences correctly for developers. The reasons can be twofold. First, the growing number of libraries and APIs result in the consequence that developers fail to fully understand them with insufficient documents as studied by Dekel and Herbsleb [15]. Second, the constraints between APIs become more and more complicated due to the increasing complexity of modern software systems and libraries. E.g., some APIs have to be used together [23, 24] while some APIs cannot [14] and even the invocation ordering of APIs matters [47]. The results provide useful insights for both practitioners and researchers: (1) API vendors are highly encouraged to provide sufficient documentation regarding API constraints, (2) precise API-constraint detection/recommendation techniques are becoming more and more desirable.

4.3 How Do API Misuses Distribute Among Projects?

Besides analyzing the *vertical* distribution of misuse categories over time, we further analyze the *horizontal* distribution of misuse categories among different projects. Typically, a misuse category spreading over more projects deserves more attention from the practitioners and developers. The treemap chart in Figure 8 shows the distribution results. In this figure, blocks with same color represent misuses in same categories such as missing API call, redundant condition and replaced APIs. Each block with same color has also several small sub-blocks which represent different sub-categories,

such as missing NULL checks in condition misuses. The area denotes the ratio of projects where corresponding misuses reside in. For clarity, we also include the concrete ratio in each block. For example, in total, there are 41.6% projects have bug fixes with the type of replaced receiver. Please note that one project can include multiple misuse categories. From the results, we have following findings. First, consistent with the raw-count results in Table 1, replaced API misuses account for almost most projects. This finding again demonstrates that replaced API misuses can not be overlooked by developers and researchers. Also, replaced receiver API misuses are still most prevalent in different projects, showing that developers tend to misuse APIs in different classes. Second, Figure 8 shows that condition-missing misuses are in more projects than API-redundant ones. However, this is not the case in Table 1 where there are clearly more API-redundant misuses. This finding illustrates that missing condition misuses are more widespread but API-redundant misuses occur intensively in a smaller number of projects. One potential reason is that APIs with constraints tend to be intensively used by certain types of projects while all projects may use APIs requiring conditional checks.

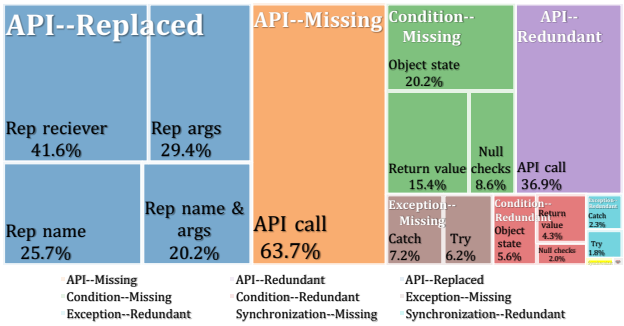

Figure 8: Treemap of bug distribution among projects.

4.4 What Are The Most Frequent Five Misuse Patterns of Different Categories?

The last three sections have performed quantitative analysis on API misuses. In this research question, we further qualitatively analyze the misuse patterns mined from our GitHub dataset. We first extract a ranked list for each misuse category according to the cross-project frequencies. The reason we consider cross-project frequencies is that the mined patterns should be helpful in detecting unknown misuses in various new projects. We remove patterns related to `printing` and `logging` because they are usually for debugging and maintenance purposes; we also remove APIs with "Android" and "Javax" since we target general Java programs. Table 2 presents the popular misuse patterns mined from each general category. Column 1 presents the name for each category. Column 2 presents the top-5 popular patterns for each category. Column 3 and 4 show the number of projects in which the corresponding pattern appeared and the total number of pattern occurrence, respectively. From this table we have following interesting findings.

First, API misuses related with class `Java.lang.String` account for the vast majority of all misuse patterns. For example, 31 out of 45 misuses in all categories are from `String` or accept `String` as arguments. Specifically, for both missing and redundant condition misuses, all top-5 API patterns are related to `String`. This finding provides practical guidance that developers have to pay more attention when they deal with APIs with `String`.

Second, all categories of replaced/missing APIs and missing exception include misuses related to the API `String.equals`. This finding shows that even though `equals` is the most basic API in Java, developers still tend to use it incorrectly. For example, developers tend to forget to check if the String is NULL when they call `equals`, incurring `NullPointerException`. In this case, `Object.equals(String,String)` is a safe way to compare two Strings.

Third, we also observe that it is totally possible to design an automated technique based on the mined patterns to detect previously unknown misuses in other projects. For example, replacing `File.mkdir()` with `File.mkdirs()` also commonly exist in Apache projects, such as the example shown in Figure 11.

5 AUTOMATIC API-MISUSE DETECTION

The results in Section 4 show that API misuses increase year by year and are widespread across many projects in history. However, whether the misuses mined from history data set are still easy to be introduced by developers in most recent projects is not evaluated. Therefore, in this paper, we propose a rule-based API-misuse detection approach to detect real-world API misuses.

For the misuse patterns also covered by existing techniques, we use the recently widely used benchmark MuBench [12] and manually analyze how many patterns in the benchmark can be found in our mined patterns. Following previous studies [13, 49], we use 53 misuses to explore the recall of our patterns on MuBench. For the misuse patterns that existing techniques cannot handle, we encode our mined patterns in the practical tool named BiD^3 and investigate how BiD^3 perform in real-world API-misuse detection. We directly apply BiD^3 to detect previously unknown misuses from the Apache projects [1], since it is widely used in both academia and practice, and has high code quality.

5.1 Misuse Detection on MuBench

In this section, we present the potential recall of misuse detection on MuBench dataset with the patterns mined from our dataset. We manually analyze the fix patterns in the MuBench and check whether the same patterns exist or not in our studied dataset. We also include the results of state-of-the-art misuse detection approaches for comparison. We directly reuse the result of MutAPI [49], while for the other techniques, i.e., DMMC [32], Jadet [48], Tikanga [47] and GrouMiner [35], the results were reported in a previous study [13]. The results are shown in Figure 9. Especially, no misuses were detected by GrouMiner. From the figure we can find that by mining from the historical bug fixes, 22 out of 53 misuses can be potentially detected, which includes 12 out of 32 distinct APIs. Additionally, from the figure we can also see that 7 out of 22 misuses are not detected by any existing techniques. Surprisingly, all 7 misuses are distinct patterns in the complete MuBench dataset, indicating that it is really hard to identify their correct usages from existing source code that was employed by all existing techniques.

Table 2: Top-5 frequently appeared fixing patterns for each categories in our dataset.

Category	Pattern	#Proj Occur	#Occur	Category	Pattern	#Proj Occur	#Occur
Exception–Redundant	`File.getParentFile()`	179	248	Condition–Redundant	`String.substring(int)`	1304	2051
	`Class.forName(String)`	146	219		`String.contains(String)`	661	1884
	`PrintStream.println(String)`	121	341		`String.equalsIgnoreCase(String)`	647	3031
	`Exception.getMessage()`	72	109		`List<String>.add(String)`	600	1070
	`StringBuffer.toString()`	70	87		`String.replace(String,String)`	579	1051
Synchronization–Redundant	`Thread.notify()`	7	9	Exception–Missing	`String.equals(String)`	2511	10197
	`Object.wait(long)`	6	9		`String.split(String)`	1909	3362
	*`Database.create(`†`ProcessInstanceImpl)`	3	6		`Thread.sleep(int)`	513	707
	`org.exolab.castor.jdo.Database.begin()`	3	6		`String.replaceAll(String,String)`	389	771
	`org.exolab.castor.jdo.Database.commit()`	3	6		`regex.Pattern.matcher(String)`	362	584
Condition–Missing	`String.substring(int,int)`	2415	4238	Synchronization–Missing	`Object.notify()`	28	31
	`String.length()`	1745	3427		‡`Configuration.getString(String)`	7	9
	`Integer.parseInt(String)`	1686	3302		`StringBuffer.setLength(int)`	6	6
	`String.trim()`	746	1451		`Map<String,Long>.remove(String)`	5	8
	`String.indexOf(String)`	744	1480		`java.nio.channels.Selector.keys()`	4	5
API–Redundant	`StringBuilder.append(String)`	798	2136	API–Missing	`StringBuilder.append(String)`	2020	5683
	`Thread.start()`	384	453		`String.equals(String)`	1194	1771
	`List<String>.add(String)`	335	884		`String.trim()`	1130	1954
	`System.exit(int)`	334	462		`List<String>.add(String)`	992	2677
	`ArrayList<String>.add(String)`	323	1221		`Thread.start()`	952	1170
API–Replaced	`String.equals(String)=>String.equalsIgnoreCase(String)`	274	655				
	`Scanner.next()=>Scanner.nextLine()`	220	601				
	`String.equals(String)=>String.contains(String)`	202	526				
	`StringBuffer.toString()=>StringBuilder.toString()`	198	375				
	`Integer.parseInt(String)=>Double.parseDouble(String)`	174	416				

In this table, we omit all the commonly used **package** declaration for clarity, such as `java.lang`, `java.util` and `java.io`.
*: denoting in the package of `org.exolab.castor.jdo`, †: denoting in the package of `engine.instance`, ‡: denoting in the package of `org.apache.commons.configuration`.

As a matter of fact, some of these fixed patterns are prevalent in historical bug fixes. For example, for the pattern `String.getBytes(String)` which is used to specify the charset of string, we can find thousands of instances with the same fixes. However, the number of occurrence for different patterns can vary greatly. Some of them are quite common, such as `String.getBytes(String)`, while some may be less prevalent, such as adding `try-catch` for `SortedMap.firstKey()`, which only occurred once before in the studied dataset. Nevertheless, the result shows that detecting API misuses by mining historical fixing commits has the potential to further improve existing techniques and the recent study by Nielebock et al. [37] also confirms this conclusion. Therefore, more research can be carried out on this topic.

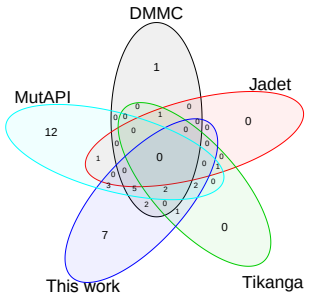

Figure 9: Overlap of bug detection results on MuBench.

5.2 Misuse Detection on Apache Projects

For detecting new unknown misuses, we select 10 replaced API-misuse patterns among top-300 from the ranking list obtained in Section 4.4. The reason we select replaced API misuses is that the detection of such misuses has not been evaluated systematically, and according to the statistical analysis in Section 4.3 it accounts for the most majority among GitHub projects. Also, since some replaced API misuses depend on the functionalities and contexts of the whole project which requires complicated program analysis, we select 10 patterns which are relatively easy to implement via AST-based pattern matching for detection. To ensure the compatibility, our technique BiD^3 still uses Gumtree [16] to match the misuse patterns with unknown APIs in new projects. We download 688 up-to-date Apache projects as our detection benchmark. Furthermore, since current static API-misuse detection tools are able to detect many misuses, we manually check and verify that the 10 misuse patterns have not been detected by existing popular detectors Findbugs [2], Infer [7] and SpotBugs [9].

Table 3 shows the empirical results of our misuse detection. In the table, the first column represents the patterns and the second column indicates the number of misuses reported by BiD^3. Since many potential misuses can be detected, we have to extract some samples. If the number of reported misuses is less than 100, we will inspect all of them, otherwise we sample only 100 misuses from the detected ones in Column 2. Column 3 shows the number of cases we sample. Furthermore, two authors manually inspect how many sampled misuses are real bugs. Column 4 represents the number we confirmed after resolving the disagreement. Column 5 indicates the number we submit to developers and Column 6 represents the number of misuses that have been confirmed and merged by the developers before the paper submission, while the last Column denotes the number of pull requests that are rejected by the developers. From the table we can find that more than half reported bugs are real bugs, indicating the effectiveness of the proposed technique. Besides, BiD^3 is currently an initial approach simply based on the insights learnt from the empirical study, which potentially has much room for improvements, such as combining machine learning [36] techniques to characterize more context features, etc. Next, we will introduce some misuse patterns and corresponding misuses we detect in real Apache projects.

`JSONObject.getString(String)=>JSONObject.optString(String)`. This pattern shows that developers ignore the `JSONException` thrown by `JSONObject.getString` if the key doesn't exist. To avoid `JSONException`, the API `optString` can be used since it will return an empty string even the queried key does not exist. To detect this kind of misuse, we scan all `JSONObject.getString` and warn if there is no handling to capture this kind of exception. Finally, 17 usages are reported as potential misuses and we submit 13 to developers after two authors'

Table 3: Detected API misuses and the feedback of submitted pull requests.

Pattern	Reported	Sampled	Confirmed	Submitted	Accepted	Rejected
JSONObject.getString(String)=>JSONObject.optString(String)	17	17	13	13	3	0
JSONObject.getJSONArray(String)=>JSONObject.optJSONArray(String)	6	6	3	2	0	0
JSONObject.getJSONObject(String)=>JSONObject.optJSONObject(String)	9	9	1	1	1	0
java.io.File.mkdir()=>java.io.File.mkdirs()	16	16	10	10	6	4
String.replaceAll(String,String)=>String.replace(String,String)	1,798	100	87	46	16	8
java.sql.Connection.createStatement()=>java.sql.Connection.prepareStatement(String)	70	70	9	9	0	0
concurrent.Executors.newCachedThreadPool()=>concurrent.Executors.newFixedThreadPool(int)	9	9	4	3	1	0
Date.getTime()=>java.lang.System.currentTimeMillis()	339	100	99	20	10	3
java.io.FileWriter.close()=>java.io.BufferedWriter.close()	74	74	61	39	20	0
String.equals(String)=>Objects.equals(String,String)	5,203	100	73	6	0	0
Total	7,541	501	360	149	57	15

```
private void mxxResult(...){
    ...
--- updatexxHit(updated,hit.getString("_id"),...);
+++ updatexxHit(updated,hit.optString("_id"),...);
    ...
    if(hitsObject.getInt("total")>currentOffset){
---   mxxResult(...,rJSON.getString("_scroll_id"));
+++   mxxResult(...,rJSON.optString("_scroll_id"));
        ...
}
```

Figure 10: Example of accepted `JSONObject.getString` bug

```
void processCommandLineArgs(String[] cliArgs) {
    ...
    File zkDir=new File(String.format("%s/zklog-parsed",
         System.getProperty("user.home")));
    if (!zkDir.exists()) {
        LOG.info("DIR"+zkDir.getAbsolutePath());
---     zkDir.mkdir();
+++     zkDir.mkdirs();
        ...
    }
}
```

Figure 11: Example of accepted `File.mkdir()` bug

confirmation. Till the paper submission, 3 pull requests are confirmed and merged into code base by developers. Figure 10 shows the one accepted misuse in project *Apache Unomi*. In this example, the queried keys may not exist and exceptions will be thrown and crash the program. As a result, the developers merged this pull request immediately without any question.

`java.io.File.mkdir()=>java.io.File.mkdirs()`. Both `File.mkdir()` and `File.mkdirs()` are used to create a directory and they all return a Boolean value. However, `mkdir()` will fail the creation if nested directories need to be created in the same time and then return false. If the demanded directory is not created, following code may cause unexpected issues even crash the system. On the other hand, `mkdirs()` can create the parent directory recursively no matter what the directory name is, even the parent does not exist. To detect this kind of misuse, we focus on the directory name of `File` object to check the value of the directory name. If the directory name includes `File.separator` or "/", it indicates that this is a nested directory. In this case, if the `File` object calls `mkdir` and fails to create the directory without any error or exception, the debugging process will be hard. Finally, 16 potential misuses are reported to us and we confirm 10 to submit to developers. Currently, 6 of them have been accepted. Figure 11 shows one accepted misuse in project *Apache Helix*.

`Executors.newCachedThreadPool()=>Executors.newFixedThreadPool(int)`. Both APIs can create a thread pool in multiple-thread environment. However, `newCachedThreadPool()` has no bounded thread number and

```
private void runBenchmarkTasks() throws Exception {
    ...
--- exc=Executors.newCachedThreadPool();
+++ exc=Executors.newFixedThreadPool(tasks.size());
    ...
}
```

Figure 12: Example of accepted `newCachedThreadPool()` bug

`newFixedThreadPool(int)` can set the maximal thread number through the arguments. In this case, `newCachedThreadPool()` may consume more and more memory if it is not constrained and the system will risk in crashing and throwing `OutOfMemoryException`. To detect this kind of misuse, we focus on the cases that executor created from `newCachedThreadPool()` submit tasks in a loop without constraints. Figure 12 shows one accepted misuse in project *Apache bookkeeper*.

The fact that 57 misuses detected by BiD[3] were accepted by developers demonstrates that our misuse pattern mining method can truly find many new and unknown API misuses, which can be complimentary to current static bug detection tools to detect more potential API misuses.

6 THREATS TO VALIDITY

The threats to external validity lie in the dataset used. To collect massive data for analysis, we mined bug-fixing commits from GitHub repositories. The dataset may have many noises (i.e., not real bug-fixing commits). The threats to internal validity relate to our implementation. To reduce errors, we use GumTree [16] to extract fine-grained AST operations, which is widely used in previous studies [21, 22, 27]. However, we cannot get certain misuse categories such as missing NULL checks directly from Gumtree. Therefore, we revise the original Gumtree by adding detailed program analysis to map GumTree operations to our classifications. Furthermore, we carefully review all our code and experimental scripts to ensure their correctness as much as we can.

7 CONCLUSION

In this paper, we conduct an extensive empirical study on API misuses based on a massive dataset including a total of 958,368 pairs of bug-fixing commits. We use fine-grained source code differencing [16] to compute edit operations of AST for collected pair files. Our empirical results show that API misuses are prominent. Then, we extract 10 replaced API misuses and apply them into Apache projects to detect unknown misuses. Finally, we report 149 misuses to developers and 57 have been confirmed, providing the practical guide of mining misuse patterns from massive historical data for API-misuse detection.

REFERENCES

[1] 2019. Apache Website. https://www.apache.org/
[2] 2019. Findbugs Website. http://findbugs.sourceforge.net/
[3] 2019. Git Website. https://git-scm.com/
[4] 2019. GitHub Archive Website. https://www.gharchive.org/
[5] 2019. GitHub Commit API. https://developer.github.com/v3/repos/commits/
[6] 2019. GitHub Website. https://github.com/
[7] 2019. Infer Website. https://fbinfer.com/
[8] 2019. Jira Website. https://www.atlassian.com/software/jira/
[9] 2019. SpotBugs Website. https://spotbugs.github.io/
[10] 2019. svn Website. https://subversion.apache.org/
[11] Mithun Acharya and Tao Xie. 2009. Mining API error-handling specifications from source code. In *International Conference on Fundamental Approaches to Software Engineering*. Springer, 370–384.
[12] Sven Amann, Sarah Nadi, Hoan A Nguyen, Tien N Nguyen, and Mira Mezini. 2016. MUBench: a benchmark for API-misuse detectors. In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*. IEEE, 464–467.
[13] Sven Amann, Hoan Anh Nguyen, Sarah Nadi, Tien N Nguyen, and Mira Mezini. 2018. A Systematic Evaluation of Static API-Misuse Detectors. *IEEE Transactions on Software Engineering* (2018).
[14] Pan Bian, Bin Liang, Wenchang Shi, Jianjun Huang, and Yan Cai. 2018. NAR-miner: Discovering Negative Association Rules from Code for Bug Detection. In *ESEC/FSE*. ACM, 411–422. https://doi.org/10.1145/3236024.3236032
[15] U. Dekel and J. D. Herbsleb. 2009. Improving API documentation usability with knowledge pushing. In *2009 IEEE 31st International Conference on Software Engineering*. 320–330. https://doi.org/10.1109/ICSE.2009.5070532
[16] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. ACM, 313–324.
[17] A. Hajnal and I. Forgacs. 2002. A precise demand-driven definition-use chaining algorithm. In *Proceedings of the Sixth European Conference on Software Maintenance and Reengineering*. 77–86. https://doi.org/10.1109/CSMR.2002.995792
[18] Quinn Hanam, Fernando S. de M. Brito, and Ali Mesbah. 2016. Discovering Bug Patterns in JavaScript. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. ACM, New York, NY, USA, 144–156. https://doi.org/10.1145/2950290.2950308
[19] Mary Jean Harrold and Mary Lou Soffa. 1994. Efficient Computation of Interprocedural Definition-use Chains. *ACM Trans. Program. Lang. Syst.* 2 (1994), 175–204. https://doi.org/10.1145/174662.174663
[20] David Hovemeyer and William Pugh. 2007. Finding more null pointer bugs, but not too many. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. ACM, 9–14.
[21] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. 2018. Shaping Program Repair Space with Existing Patches and Similar Code. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 298–309. https://doi.org/10.1145/3213846.3213871
[22] X. B. D. Le, D. Lo, and C. L. Goues. 2016. History Driven Program Repair. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. 213–224. https://doi.org/10.1109/SANER.2016.76
[23] Zhenmin Li and Yuanyuan Zhou. 2005. PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. In *ACM SIGSOFT Software Engineering Notes*, Vol. 30. ACM, 306–315.
[24] B. Liang, P. Bian, Y. Zhang, W. Shi, W. You, and Y. Cai. 2016. AntMiner: Mining More Bugs by Reducing Noise Interference. In *ICSE*. 333–344. https://doi.org/10.1145/2884781.2884870
[25] Guangtai Liang, Qianxiang Wang, Tao Xie, and Hong Mei. 2013. Inferring project-specific bug patterns for detecting sibling bugs. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 565–575.
[26] Christian Lindig. 2015. Mining patterns and violations using concept analysis. In *The Art and Science of Analyzing Software Data*. Elsevier, 17–38.
[27] K. Liu, D. Kim, T. F. Bissyande, S. Yoo, and Y. Le Traon. 2018. Mining Fix Patterns for FindBugs Violations. *IEEE Transactions on Software Engineering* (2018). https://doi.org/10.1109/TSE.2018.2884955
[28] Benjamin Livshits and Thomas Zimmermann. 2005. DynaMine: finding common error patterns by mining software revision histories. In *ACM SIGSOFT Software Engineering Notes*, Vol. 30. ACM, 296–305.
[29] Matias Martinez and Martin Monperrus. 2015. Mining Software Repair Models for Reasoning on the Search Space of Automated Program Fixing. *Empirical Softw. Engg.* 20, 1 (Feb. 2015), 176–205. https://doi.org/10.1007/s10664-013-9282-8
[30] Na Meng, Stefan Nagy, Danfeng (Daphne) Yao, Wenjie Zhuang, and Gustavo Arango Argoty. 2018. Secure Coding Practices in Java: Challenges and Vulnerabilities. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. ACM, New York, NY, USA, 372–383. https://doi.org/10.1145/3180155.3180201
[31] Martin Monperrus, Marcel Bruch, and Mira Mezini. 2010. Detecting missing method calls in object-oriented software. In *European Conference on Object-Oriented Programming*. Springer, 2–25.
[32] Martin Monperrus and Mira Mezini. 2013. Detecting missing method calls as violations of the majority rule. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 22, 1 (2013), 7.
[33] H. A. Nguyen, A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, and H. Rajan. 2013. A study of repetitiveness of code changes in software evolution. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 180–190. https://doi.org/10.1109/ASE.2013.6693078
[34] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H Pham, Jafar Al-Kofahi, and Tien N Nguyen. 2010. Recurring bug fixes in object-oriented programs. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering-Volume 1*. ACM, 315–324.
[35] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, Jafar M. Al-Kofahi, and Tien N. Nguyen. 2009. Graph-based Mining of Multiple Object Usage Patterns. In *ESEC/FSE*. ACM, New York, NY, USA, 383–392. http://doi.acm.org/10.1145/1595696.1595767
[36] Tam The Nguyen, Hung Viet Pham, Phong Minh Vu, and Tung Thanh Nguyen. 2015. Recommending API usages for mobile apps with hidden markov model. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 795–800.
[37] Sebastian Nielebock, Robert Heumüller, and Frank Ortmeier. 2018. Commits As a Basis for API Misuse Detection. In *Proceedings of the 7th International Workshop on Software Mining (SoftwareMining 2018)*. ACM, New York, NY, USA, 20–23. https://doi.org/10.1145/3242887.3242890
[38] Kai Pan, Sunghun Kim, and E. James Whitehead, Jr. 2009. Toward an Understanding of Bug Fix Patterns. *Empirical Softw. Engg.* 14, 3 (2009), 286–315. http://dx.doi.org/10.1007/s10664-008-9077-5
[39] M. K. Ramanathan, A. Grama, and S. Jagannathan. 2007. Path-Sensitive Inference of Function Precedence Protocols. In *29th International Conference on Software Engineering (ICSE'07)*. 240–250. https://doi.org/10.1109/ICSE.2007.63
[40] Murali Krishna Ramanathan, Ananth Grama, and Suresh Jagannathan. 2007. Static specification inference using predicate mining. In *ACM SIGPLAN Notices*, Vol. 42. ACM, 123–134.
[41] Baishakhi Ray, Vincent Hellendoorn, Saheel Godhane, Zhaopeng Tu, Alberto Bacchelli, and Premkumar Devanbu. 2016. On the "Naturalness" of Buggy Code. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, 428–439. http://doi.acm.org/10.1145/2884781.2884848
[42] M. Soto, F. Thung, C. Wong, C. Le Goues, and D. Lo. 2016. A Deeper Look into Bug Fixes: Patterns, Replacements, Deletions, and Additions. In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*. 512–515. https://doi.org/10.1109/MSR.2016.067
[43] Suresh Thummalapenta and Tao Xie. 2009. Alattin: Mining alternative patterns for detecting neglected conditions. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 283–294.
[44] Suresh Thummalapenta and Tao Xie. 2009. Mining exception-handling rules as sequence association rules. In *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 496–506.
[45] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2018. An empirical investigation into learning bug-fixing patches in the wild via neural machine translation. In *ASE*. 832–837.
[46] Z. Wan, D. Lo, X. Xia, and L. Cai. 2017. Bug Characteristics in Blockchain Systems: A Large-Scale Empirical Study. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. 413–424. https://doi.org/10.1109/MSR.2017.59
[47] Andrzej Wasylkowski and Andreas Zeller. 2009. Mining Temporal Specifications from Object Usage. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering (ASE '09)*. 295–306. https://doi.org/10.1109/ASE.2009.30
[48] Andrzej Wasylkowski, Andreas Zeller, and Christian Lindig. 2007. Detecting object usage anomalies. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM, 35–44.
[49] Ming Wen, Yepang Liu, Rongxin Wu, Xuan Xie, Shing-Chi Cheung, and Zhendong Su. [n.d.]. Exposing Library API Misuses via Mutation Analysis. ([n. d.]).
[50] Chadd C Williams and Jeffrey K Hollingsworth. 2005. Automatic mining of source code repositories to improve bug finding techniques. *IEEE Transactions on Software Engineering* 31, 6 (2005), 466–480.
[51] R. Yue, N. Meng, and Q. Wang. 2017. A Characterization Study of Repeated Bug Fixes. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 422–432. https://doi.org/10.1109/ICSME.2017.16
[52] Hao Zhong and Zhendong Su. 2015. An Empirical Study on Real Bug Fixes. In *Proceedings of the 37th International Conference on Software Engineering (ICSE '15)*. 913–923. http://dl.acm.org/citation.cfm?id=2818754.2818864