A Large-scale Study on API Misuses in the Wild

Xia Li
Department of Software Engineering and Game Design, Kennesaw State University
xli37@kennesaw.edu

Jiajun Jiang†
College of Intelligence and Computing, Tianjin University
jiangjiajun@tju.edu.cn

Samuel Benton
Department of Computer Science, The University of Texas at Dallas
Samuel.Benton1@utdallas.edu

Yingfei Xiong
Key Laboratory of High Confidence Software Technologies (MoE); DCST, Peking University
xiongyf@pku.edu.cn

Lingming Zhang
Department of Computer Science, University of Illinois at Urbana-Champaign
lingming@illinois.edu

***Abstract*—API misuses are prevalent and extremely harmful. Despite various 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 large-scale empirical study on API misuses based on 528,546 historical bug-fixing commits from GitHub (from 2011 to 2018). By leveraging a state-of-the-art fine-grained AST differencing tool, GumTree, we extract more than one million bug-fixing edit operations, 51.7% of which are API misuses. We further systematically classify API misuses into nine different categories according to the edit operations and context. We also extract various frequent API-misuse patterns based on the categories and corresponding operations, which can be complementary to existing API-misuse detection tools. Our study reveals various practical guidelines regarding the importance of different types of API misuses. Furthermore, based on our dataset, we perform a user study to manually analyze the usage constraints of 10 patterns to explore whether the mined patterns can guide the design of future API-misuse detection tools. Specifically, we find that 7,541 potential misuses still exist in latest Apache projects and 149 of them have been reported to developers. To date, 57 have already been confirmed and fixed (with 15 rejected misuses correspondingly). The results indicate the importance of studying historical API misuses and the promising future of employing our mined patterns for detecting unknown API misuses.**

***Index Terms*—Pattern generation, Program adaptation, Code abstraction**

I. INTRODUCTION

Over the past decades, software systems have been widely used 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 significant 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 [1], [2]. In Java, for example, the correct way to create a new thread is to call API `Thread.start()`, while `Thread.run()` is often misused, which does not create new threads. 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 potentially obtained from historical bug fixes.

A large number of historical bug fixes are publicly available on open-source platforms (e.g., GitHub [3] and SourceForge [4]) and issue tracking systems (e.g., JIRA [5]). Many researchers have conducted various studies to analyze bug fixes and have many findings. For example, Martinez and Monperrus [6] found that most 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 researchers have studied historical bug fixes, they only examined a small dataset, and the findings could be limited. In this paper, we conduct a study on API misuses by analyzing much more comprehensive bug-fixing dataset than any prior work.

Many techniques define API misuses as violations of certain frequent API usage patterns which are mined from source code [7]–[11]. These techniques were evaluated to be effective on some kinds of misuses. However, a recent study on state-of-the-art API-misuse detectors [1] shows that API misuses are prevalent and existing API-misuse detectors suffer from extremely low precision and recall on the 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 various API misuses but also incurs dataset overfitting issues 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 by analyzing a large-scale bug-fixing dataset. We start with mining the historical repositories from GitHub Archive [13] that records the public GitHub timeline dataset. Then, we extract all bug-fixing commits of Java projects from 2011 to 2018 according to specific search criteria, resulting in 528,546 bug-fixing commits. Finally, we extract fine-grained edit operations on AST

† Corresponding author.

(Abstract Syntax Tree) of buggy and fixed source code via leveraging GumTree [14] to identify API misuses. Following existing studies [1], we classify API misuses into different categories and statistically analyze their distributions. From the statistical results, we observe various practical guidelines regarding the importance of different categories of API misuses. For example, API calls (missing/redundant) are the most frequent (34.05% of all API misuses) and synchronization is the least frequent (0.06% of all API misuses). In addition, we find that about 38.33% API misuses are replacing API components (i.e., arguments, receiver, or name), which have never been systematically explored by any prior studies.

Inspired by the statistical results, we extract a large set of frequent API-misuse patterns for each category, and carry out two studies on API-misuse detection. (1) API-misuse detection in MUBENCH. Following previous studies [1], [15], we analyze how many API-misuse patterns in MUBENCH can also 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 patterns in MUBENCH have already occurred before, and 7 of them cannot be detected by any existing techniques, demonstrating the possibility to build more effective misuse detection tools via analyzing large-scale historical bug fixes. (2) API-misuse detection in the wild. Based on our dataset, we perform a user study to manually analyze the usage constraints of 10 representative API-misuse patterns to explore whether the mined patterns can guide the design of future API-misuse detection tools. The results of the experiments on 688 Apache projects show that the misuses extracted from historical records still exist in the latest projects (7,541 potential API misuses in total). We have reported 149 misuses to developers. Up to now, 57 of them have already been confirmed and fixed while 15 are rejected. The promising initial results indicate that the misuse patterns complement existing approaches and can potentially improve the practicability of future API-misuse detection techniques.

In summary, this paper makes the following contributions:

- **Dataset.** A publicly available dataset including 528,546 historical bug-fixing commits from GitHub (from 2011 to 2018).
- **Study.** An extensive study to analyze API misuses in the wild and a systematic classification of them via static analysis.
- **Patterns.** A large set of API-misuse patterns, which complement existing API-misuse detection tools and can be used for detecting unknown API misuses in the wild.
- **Guidelines.** Various practical guidelines regarding the importance of different types of API misuses, such as (1) among the API misuses covered by prior techniques, API call is the most frequent (34.05%) while synchronization is the least frequent (0.06%), (2) Replaced API misuses, not covered by any prior techniques, account for the largest portion among all misuses (38.33%), (3) misuse patterns related to class `java.lang.String` account for the vast majority of all patterns, and (4) most frequent Replaced API-misuse patterns are related to APIs in the same class.
- **Application.** Finally, we explore whether the dataset and patterns can be applied to API-misuse detection by leveraging 10 patterns and manually-defined heuristic rules after analyzing the corresponding repair histories in our dataset. The experimental results show that misuses extracted from historical records still exist in the latest versions of Apache projects (149 of them are reported to developers, 57 have already been confirmed and fixed, and 15 are rejected).

II. 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.

A. Analysis of Historical Bug Fixes

During modern software development, a large number of historical bug fixes get accumulated on open-source platforms (e.g., GitHub and SourceForge) and issue tracking systems (e.g., JIRA). Understanding and analyzing the historical bug fixes can potentially provide practical guidelines for manual or automated bug detection [16]–[19], localization [20]–[25], and repair [26]–[30]. Therefore, various studies have been conducted on historical bug fixes. Zhong and Su [31] 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. Additionally, they found that most bug fixes are related to `if` conditions, which is also confirmed by Soto et al. [32]. Pan et al. [33] 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. Recently, the benchmark Defects4J [34] which includes various real bugs has been widely used in the field of software debugging. Sobreira et al. [35] studied 395 bugs in Defects4J and found that the top-3 most applied bug-fixing actions (77% of the total bugs) are addition of method calls, conditionals, and assignments. Martinez et al. and Madeiral et al. proposed different bug detection tools (Coming [36] and PPD [37] respectively) and evaluated them on Defects4J benchmark with promising performance.

Besides, there were also studies on domain-specific bug patterns or bug pattern distribution across different projects. Wan et al. [38] used the card-sorting approach to analyze the characteristics of bugs in Blockchain systems. Meng et al. [39] 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. [40] 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. [41] and Nguyen et al. [42]. Ray et al. [43] found that although source code is highly

repetitive and predictable (like natural languages), the buggy code tends to be unnatural. After comparing with different statistical 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, we aim to perform an extensive 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 528,546 historical bug-fixing commits from open-source projects and represents the most extensive study on historical bug fixes to date.

B. Studies on API-misuse Detection

API usage is often subject to certain constraints [1]. For example, a resource must be released or closed after it is used. Violations of such usage patterns are regarded as API misuses. A large number of techniques have been proposed to detect API misuses automatically over the past decades.

Most approaches utilize data mining techniques to detect API misuses. Livshits et al. [44] introduced DynaMiner, aiming to mine software revision histories to detect misuses violating method pairs or certain mined state machines. Similarly, Li and Zhou [45] proposed PR-Miner to extract implicit programming rules of APIs by leveraging frequent itemset mining approaches on source code. Acharya and Xie [9] proposed to mine specifications from static program traces. Although such techniques utilize different data sources, they share the same assumption that the more frequent a pattern is, the higher possibility a pattern is correct. Other similar techniques also include DMMC [2], [46], GrouMiner [47], COLIBRI/ML [7], etc. Additionally, Wasylkowski et al. [8] and Nguyen et al. [48] 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. [19] and Hovemeyer et al. [18] targeted missing NULL pointer checks, while Thummalapenta and Xie focused on exception-handling related misuses [10] and neglected-condition misuses [11]. More recently, Liang et al. [17] aimed at detecting 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.

Besides data mining, researchers also employed program analysis and machine learning for API-misuse detection. Ramanathan et al. proposed CHRONICLER [49] and RGJ07 [50], utilizing path-sensitive control-flow or data-flow analysis to infer function precedence protocols or predicates. Wasylkowski and Zeller [51] proposed TIKANGA to combine static analysis with model checking for mining Computation Tree Logic (CTL) formulas. Nguyen et al. [52] leveraged Hidden Markov Model to check anomalies of call sequences. Most recently, Wen et al. [15] applied mutation analysis to discover API misuse patterns to improve the state-of-the-art.

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. Whether the existing techniques have covered all major types of API misuses is also not investigated. 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 much larger dataset for API-misuse detection.

III. EMPIRICAL STUDY

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

A. Data Collection

We aim to mine API-misuse patterns from all bug-fixing commits of Java projects on GitHub. To collect our dataset, we first download all public GitHub events for all program languages from GitHub Archive [13] 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, following prior work [53], we identify a commit as a bug fix if its commit 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"). Since the commit message may not identify bug-fixing commits accurately, we randomly select 100 commit samples, and two authors independently analyze them to check whether they are actual bug fixes. The result is that 94% of the identified bug-fixing commits are real bug fixes, which provides us more confidence for the subsequent analysis. We also keep only unique commits by removing duplicates. Next, we download the source files before and after the code change for each bug-fixing commit. To mitigate the impact of irrelevant code changes, we discard commits or files that meet any of the following criteria.

- Commits with changes involving more than five Java files or six lines of source code [53], [54], since such commits may include many changes not related to bug fixes.
- Non-Java files as they are irrelevant to Java API misuses.
- Java files that deleted or newly introduced in the commits.

As a result, we finally get 528,546 bug-fixing commits (including 220,053 projects and 744,000 pairs of buggy and fixed files) for further API-misuse pattern mining.

B. Categorization of API Misuses

Following prior study [1], we define an *API misuse* as a pair of a violation type and an API-usage element (e.g., API call, iteration, condition, and exception handling) involved in a bug

fix. Besides the *missing* and *redundant* violation types studied in prior study [1], in this paper, we further investigate the type of *replaced*, describing that an API is incorrectly invoked and should be replaced with another one. This type of API misuses has never been systematically studied before, but is prevalent in real-world projects (will be shown in the following sections). In total, we classify API misuses into four basic categories, including Condition, Exception, Synchronization and API call, each of which consists of some specific sub-categories. In the following, we demonstrate each category of API misuses in detail.

Condition. This category includes missing and redundant guard conditions for certain API invocations. Following the previous study [1], we further categorize it into the following three sub-categories:

- *NULL checks.* This sub-category indicates removing or newly introducing an `if` condition with NULL checks for the variable that is returned by a prior API call or will be 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 that 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 relates to some variables that will be used in an API call immediately, e.g., `a = list.get(i); => if(i > 0){a = list.get(i);}`.

Importantly, the 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 III-C.

Exception. This category includes missing and redundant exception handlers, following the definitions in the prior work [1]. More specifically, we further divide this category into two types of fine-grained code changes, i.e., inserting or deleting `Try` or `Catch` blocks. The reason is that in the studied commits, we find 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 irrelevant. Besides, like the **Condition** category introduced above, 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 subjects to addition or deletion of `catch` blocks, whose corresponding `try` blocks contain API invocations.

Synchronization. This category includes missing and redundant synchronizations in multi-threaded environments, following the previous study by Amann et al. [1]. The difference is that we classify this type of code changes as an independent one rather than a sub-category of Condition.

```
// 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);
```

Fig. 1: Examples of replaced bugs

API call. Previous studies have focused on missing [45], [46] and redundant [52] API call misuses. However, more fine-grained API changes (such as changing only the arguments, names or receiver objects of API invocations) were not been systematically and extensively studied by existing studies. In this paper, besides missing and redundant API changes, we further investigate the distributions of replaced API misuses, which include four categories in detail. For the missing and redundant API misuses, previous study [1] already introduced them (a.k.a. Method Call). To make the article self-contained, we redundantly explain them 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 pairwise APIs that have usage dependency. Similarly, *Redundant API call* represents that an 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 API has side-effects, whose execution may conflict with the followed 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 first example shown in Figure 1, where the desired API is `nextInt()` without arguments. On the contrary, a wrong API `nextInt(int)` is 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 (order matters), while it is not our cases to change the referred object of same types. 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. Therefore, a complex points-to analysis [55] is not needed, and the built-in type analysis of GumTree will be sufficient.

- *Replaced name.* This category implies that developers call a wrong API with exactly the same arguments but different API names from the same class. This kind of misuses is usually caused by the similarity or confusion of different API names. Figure 1 shows one such 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 both the name and the arguments are incorrectly used for a method call, 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 clause, `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 1).
- *Replaced receiver.* The above three replaced API misuses are all related to misuses of APIs from the same class. However, sometimes developers even misuse APIs belonging to different classes. We classify this kind of replacements as *replaced receiver*. As introduced above, we only consider the type change of receiver. For example, the last code change in Figure 1 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.

In the study, 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 described in the documents. It is easy to identify this API misuse by manual inspection [1], while it will be infeasible in our study on such a large dataset, because it is hard to automatically analyze such specifications of all API usages from informal documents that are usually presented via natural language.

C. Edit Operation Extraction

To study different misuses defined in Section III-B, we first extract code changes from historical bug-fixing commits and then map them into corresponding categories.

In our study, we apply a state-of-the-art AST (Abstract Syntax Tree) differencing tool GumTree [14], which can extract fine-grained AST operations and has been widely employed by previous studies [27], [30], [56]. According to the classifications, we consider three types of operations defined by it, i.e., *update*, *delete* and *insert*. Then, according to the content of changed code and the type of operation, we map AST operations to the corresponding misuse categories. Particularly, a commit may include multiple operations.

```
    public static List<String> writeFiles(State
        state,...){
        ...
---     fPath=fPath.replaceAll("/",File.separator);
    // The first arg of replaceAll()
+++     fPath=fPath.replace("/",File.separator);
    // will be treated as regex
        ...
        if (!isLoaded) {
           Log.e("WavefrontLoader","Error loading");
---         System.exit(1);  // should not terminate
     the application
        }
    }
```

Fig. 2: Examples of *update* and *delete* operations.

```
    public void setAnimationStyle(int animRes) {
        Window window = dialog.getWindow();
+++      if (window != null)
    {  // add NPE check to avoid crash
            window.setWindowAnimations(animRes);
+++      }
    }
```

Fig. 3: Examples of *insert* and *move* operations.

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 demand-driven lightweight intra-procedural data- and control-dependency analysis. More concretely, to identify whether an inserted `if` condition is checking the return value of some APIs, we need to analyze the Use-Define chain [57], [58] for variables used in the condition: the operation can be mapped to the type of "*Return value*" only if the variables are the return values of some early API invocations. Similarly, to identify the type of "*Object state*", both data and control dependencies will be utilized. As a result, for variables used as receivers or arguments of certain APIs, we will perform a backward slicing within the method. While for the variables defined by APIs, a forward slicing will be computed. In this way, a relatively small number of variables need to be analyzed, and thus it is highly efficient compared to a thorough analysis of the complete program. Finally, we combine the results of GumTree and our lightweight dependency analysis to determine the API-misuse categories of bug-fixing operations.

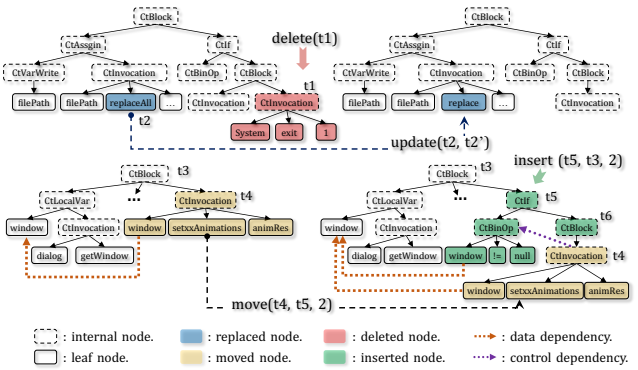

Fig. 4: GumTree operations of examples in Figures 2-3.

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

Definition 1. An abstract syntax tree (AST) is a partial ordered 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 nodes in the subtree, it will be $\emptyset$ for leaf nodes.

Finally, based on the description of operations in GumTree [14], it will be straightforward to give the operation definitions under the AST definition.

Definition 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'.

Particularly, we discard all *move* operations in the mapping process as it mainly changes the code structure but not the API itself, which is hard to be automatically analyzed as misuses. Next, we use the operations shown in Figure 4 as examples to demonstrate the mapping process in detail. According to the explanation for each category of API misuses in Section III-B, the operation $delete(t1)$ will be classified as *redundant* API call, while the operation $update(t2, t2')$ will be classified as *Replaced name* of API call. In particular, when the name and arguments of a method call are changed together, the operations will be combined as one *Replaced name and arguments* (e.g., updating `createStatement()` to `prepareStatement(String)` in Figure 1). For the operation $move(t4, t5, 2)$, we simply ignore it and in fact it does not misuse any API. Finally, as for the operation of $insert(t5, t3, 2)$, from the figure we can see that a NULL check condition for the variable `window` is inserted, which is the returned value of API `getWindow()`. As a consequence, it will be classified as missing both *NULL checks* and *Return value*. Additionally, variable `window` is further used by API `setWindowAnimations()` that has control-dependency on the condition w.r.t. `window`. As a result, it will be classified as *Object state* as well. Thus, one operation may be classified into multiple categories in *Condition*. Based on this process, we automatically classify GumTree operations into different categories for further analysis.

IV. Empirical Result Analysis

According to the previous sections, we collect a large number of API misuses in real-world projects. In this section, we conduct various empirical studies and discuss the results.

A. Distributions of API Misuses

In this research question, we count the number of edit operations from GumTree for API misuses and non-API misuses, and then analyze the distribution of different categories of API misuses. The result shows that there are 576,515 studied operations involving API misuses, which is about 51.7% of all edit operations (i.e., 48.3% for non-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 third-party libraries to save development efforts and improve code quality. To our best knowledge, this is the first study quantitatively demonstrating the importance of API-misuse in modern software systems.

TABLE I: Distribution of API misuses

Category		Missing	Redundant
API call		142,206 (24.67%)	54,101 (9.38%)
Synchronization		308 (0.05%)	58 (0.01%)
Condition	NULL checks	11,750 (2.04%)	1,320 (0.23%)
	Return value	21,900 (3.80%)	3,162 (0.55%)
	Object state	29,873 (5.18%)	4,330 (0.75%)
	Total	63,523 (11.02%)	8,812 (1.53%)
Exception	Try	6,118 (1.06%)	790 (0.14%)
	Catch	7,183 (1.25%)	1,152 (0.20%)
	Total	13,301 (2.31%)	1,942 (0.34%)
Replaced API	Rep Receiver	101,985 (17.69%)	
	Rep Name	45,963 (7.97%)	
	Rep Args	52,277 (9.07%)	
	Rep Name&Args	20,744 (3.60%)	
	Total	220,969 (38.33%)	

Table I presents the distributions of different categories of API misuses described in Section III-B. In the table, the first column represents the categories of misuses, and the last two columns represent the number of operations for *missing* and *redundant* misuses, respectively. Particularly, the percentage in the table represents the number of operations over that of all API-related operations (i.e., 576,515). As explained before, different categories may overlap each other (e.g., NULL checks and Return value). Besides, we also separately list the number and percentage of operations related to misuses of Replaced API, which is an important category in the study. In addition, for clarity, we omit the operations that are not related to API misuses as they are not the focus of this paper. From this table, we have following findings.

First, API call and Replaced API misuses are more prevalent. From the table, the percentages of operations about API call and Replaced API are more than 70%. Particularly, 38.33% misuses are about Replaced API, which are more than any other types of API misuses. By analyzing the data, we find that one important reason for such a large portion of Replaced API misuses is that most of APIs share similar signatures when their functionalities are close. Therefore, if developers do not well understand the difference between APIs, they tend to be confused and use a wrong API. For example, when one

wants to only get the miliseconds of current time, `System.currentTimeMillis()` is a preferable API with high efficiency. However, developers tend to misuse the API `new Date().getTime()`, which is simply a wrapper of the former. Due to the new `Date` object, the latter API may cause performance issues, especially when it is intensively used in time-critical programs. Therefore, if possible, it is better to directly use the API `System.currentTimeMillis()` to speed up the underlying system. The result demonstrates the importance of Replaced API misuses, and more research efforts are informed to be dedicated to detecting such misuses.

Second, missing API is more prevalent than redundant. The results show that developers tend to *miss* some API calls or handlers (such as Condition, Synchronization, and Exception) rather than writing *redundant* ones. For example, the percentage of missing API calls is 24.67%, almost three times higher than the opposite. Similarly, the percentage of missing condition is also much higher than that of redundant condition (11.02% vs 1.53%).

> **Finding 1:** (1) API call and Replaced API are the most prevalent API misuses, and Replaced API misuses accounts for the largest portion among all misuses, calling for new detection approaches. (2) Developers tend to miss some components to satisfy the constraint of a certain API.

B. Frequencies of API Misuse Patterns

In Section IV-A, we have performed quantitative analysis on API misuses. In this research question, we further qualitatively analyze the misuse patterns mined from the studied 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 more helpful in detecting unknown misuses if they widely exist in more various 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 II presents the popular misuse patterns for each general category. Column 1 denotes the category names. 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 occurrences, respectively. From this table we have following findings.

First, API misuses related to class `java.lang.String` account for the vast majority of all misuse patterns. For example, 18 out of 45 misuses in all categories are from the class of `String`. Also, for both missing API and missing condition misuses, all top-5 API patterns are related to `String`. Specifically, we have the following observations. (1) For the missing condition, most patterns miss checking if the index of a substring or a character inside a `String` is valid, such as `String.charAt(int)`. (2) There are various ways to fix bugs related to API misuses. For example, to deal with potential bugs of `String.lastIndexOf(String)`, inserting either condition or exception handling is reasonable in historical bug-fixing dataset. The findings show the importance of `String` and guide developers how to detect and fix misuses related to `String` by mining bug-fixing dataset.

Second, most Replaced API misuse patterns are related to the names and arguments in the same class, even though the Replaced Receiver is the majority in Table I. For example, developers tend to misunderstand between `Integer.valueOf(String)` and `Integer.parseInt(String)`, where the former returns an `Integer` object while the latter returns a primitive `int` value.

Third, it is possible to design an automated technique based on the mined patterns to detect unknown misuses in other projects. For example, there is a misuse pattern `File.mkdir()=>File.mkdirs()` in the ranking list. We have detected such misuses existing in Apache projects, and one submitted pull request has been accepted by developers, shown in Figure 7. In fact, there are many valuable patterns in the ranking list and we will introduce how we are inspired to improve misuse detection in the following two sections.

> **Finding 2:** (1) API misuses related to class `java.lang.String` account for the vast majority of all misuse patterns. (2) Most frequent Replaced API misuse patterns are related to the names and arguments in the same class. (3) The frequent API misuses in Table II informs new misuse detection techniques.

C. Study of API Misuse Detection on MUBENCH

In this section, we present the potential recall of misuse detection on the recently widely used benchmark MUBENCH [1], [12] with the patterns mined from our dataset. We manually analyze the fix patterns in the MUBENCH and then check whether the same patterns **exist** in the studied dataset. We assume that an ideal detection approach can accurately mine API-misuse patterns from historical fixes if at least one fix instance exists in the dataset. In addition, to explore the complementariness to existing approaches, we also include the results of state-of-the-art misuse detection approaches, including MutAPI [15], DMMC [2], Jadet [8], Tikanga [51] and GrouMiner [47].

Figure 5 presents the overlaps of misuses detected by different approaches, where "This work" denotes the results mined from our dataset. As a result, the 53 misuse examples in MUBENCH involve 32 different kinds of API-misuse patterns (multiple examples may relate to a same pattern), and 12 patterns can be found in our dataset, which correspond to 22 misuse examples. In other words, 22 misuses in MUBENCH potentially can be detected with the patterns mined from historical bug fixes. Besides, 7/12 misuse patterns cannot be detected by any existing approaches, indicating that mining misuse patterns from large-scale historical bug fixes has the potential to further improve the effectiveness of API misuse detection. For example, the API misuse of `String.getBytes()`

TABLE II: Top-5 frequently appeared fixing patterns for each category in our dataset.

Cat.	Pattern	#Proj Occur	#Occur	Cat.	Pattern	#Proj Occur	#Occur
API–Redundant	`StringBuilder.append(String)`	193	255	API–Missing	`StringBuilder.append(String)`	615	995
	`String.toLowerCase()`	92	125		`String.trim()`	458	676
	`System.exit(int)`	89	109		`String.toLowerCase()`	380	682
	`String.trim()`	87	110		`String.replace(String,String)`	305	439
	`List<String>.add(String)`	69	94		`String.replaceAll(String,String)`	294	421
Exception–Redundant	`Class.forName(String)`	44	59	Exception–Missing	`String.replaceAll(String,String)`	152	202
	`java.text.SimpleDateFormat.parse(String)`	22	27		`Thread.sleep(int)`	135	160
	`Class<?>.newInstance()`	20	34		`org.json.JSONObject.getString(String)`	112	138
	`File.isDirectory()`	17	21		`String.lastIndexOf(String)`	109	127
	§`.awaitTermination(int,concurrent.TimeUnit)`	11	12		`BufferedReader.readLine()`	88	107
Synchro–Redundant	`Object.notifyAll()`	13	18	Synchro–Missing	`Object.wait()`	7	7
	`Object.wait(long)`	4	5		`List<String>.add(int,String)`	6	6
	∗`.Database.create(`†`.ProcessInstanceImpl)`	3	6		‡`.Configuration.getString(String)`	4	5
	∗`.Database.begin()`	3	6		`Object.notify()`	4	5
	∗`.Database.commit()`	3	6		`Map<`¶`,`‖`>.get(mperm.getActionEffect())`	3	4
Condition–Redundant	`String.substring(int)`	499	590	Condition–Missing	`String.substring(int,int)`	928	1108
	`File.mkdir()`	61	65		`String.length()`	607	709
	`HashMap<String, String>.put(String,String)`	42	52		`Integer.parseInt(String)`	444	542
	`reflect.Field.getType()`	29	33		`String.indexOf(String)`	263	306
	`request.getSession().getAttribute(String)`	26	32		`String.charAt(int)`	230	254
API–Replaced	`String.split(String)=>String.split(String,int)`	89	127				
	`String.equals(String)=>String.equalsIgnoreCase(String)`	87	189				
	`Scanner.next()=>Scanner.nextLine()`	83	118				
	`Integer.valueOf(String)=>Integer.parseInt(String)`	75	124				
	`String.getBytes()=>String.getBytes(java.nio.charset.Charset)`	70	110				

In this table, we omit all the commonly used **`package`** declaration for clarity, such as `java.lang`, `java.util` and `java.io`.
∗: package of `org.exolab.castor.jdo`, †: package of `engine.instance`, ‡: package of `org.apache.commons.configuration`
§: package of `concurrent.ExecutorService`, ¶: `Action.ActionEffect`, ‖: `org.jboss.as.controller.access.permission.ManagementPermission`

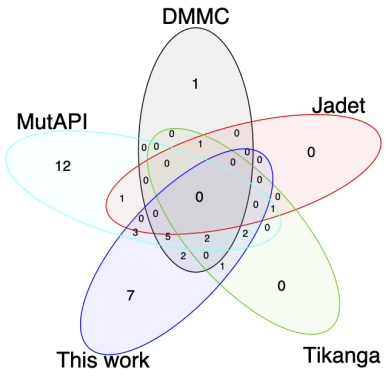

Fig. 5: Overlaps of API-misuse analysis on MUBENCH.

(should be replaced with `String.getBytes("UTF8")`) cannot be detected by any existing approaches, while thousands of bug fixes of this pattern exist in our dataset.

However, detecting API misuses based on historical fixes is not easy since some patterns are still less prevalent and hard to learn, such as adding `try-catch` for `SortedMap.firstKey()`, which only occurred once in history. Therefore, learning patterns from a small number of examples is desirable, which is also the key challenge for automatic techniques. 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. [59] also confirms this conclusion. Therefore, more research can be carried out on this topic.

Finding 3: 7 misuse patterns in MUBENCH cannot be detected by any existing approaches but exist in our dataset, indicating the potential to further improve the effectiveness of API misuse detection.

D. Study of API Misuse Detection on Apache Projects

In this section, we explore whether the dataset and patterns from previous analysis can be applied to API-misuse detection in the latest Apache projects.

Patterns. We select 10 misuse patterns of Replaced API from the ranking list obtained in Section IV-B. 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 IV-A it accounts for the most majority in history. Also, since some Replaced API misuse patterns have been studied by existing popular detectors, e.g., Findbugs [60], Infer [61] and SpotBugs [62], we select the 10 patterns that cannot been detected by any of these detectors and thus will complement them. The details of the patterns are listed in Table III.

Dataset. We use the latest Apache projects as our experimental dataset. The reasons are twofold. First, those projects are widely used by both commercial corporations and researchers, and thus the quality of projects can be relatively reliable. Second, most of them are well maintained and we potentially can get quick feedback. As a result, we download 688 Apache projects in total.

Procedure. First, we manually analyze the usage constrains and repair histories per pattern to build corresponding heuristic rules. To assist our analysis, we have implemented a simple tool to encode such heuristics via static program analysis. Then, we run the tool over the experiment dataset, and potential API misuses will be reported. Next, we randomly sample 100 reports (if exist) per pattern for manual confirmation. Two authors manually check them and discuss to reach a consensus for disagreements. Finally, we report the confirmed misuses to maintainers as many as possible.

Results. Table III shows the empirical results. In the table, the first column represents the patterns. The following columns show the number of misuses that are *reported*, *sampled* for manual inspection, *confirmed* by us, *submitted* to project maintainers, *accepted* and merged as true misuses, and *rejected* as false positives. All above data are updated until the paper submission. From the table, we can find 57 out of 149 reported instances are real misuses and 15 are rejected, indicating the potential of building more practical API-misuse detection tool via leveraging our mined patterns. However, the current tool implementation still depends on our handcrafted detection

TABLE III: 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 migrateTagsInResult(String
        hostAddress, ...){
        ...
---     updateTagsForHit(updated,hit.getString("_id")
    ,...);
+++     updateTagsForHit(updated,hit.optString("_id")
    ,...);
        ...
    if(hitsObject.getInt("total")>currentOffset){
---         migrateTagsInResult(...,rJSON.getString("
    _scroll_id"));
+++         migrateTagsInResult(...,rJSON.optString("
    _scroll_id"));
            ...
    } //https://github.com/apache/unomi/commit/
        c447224
```

Fig. 6: Accepted `JSONObject.getString()` misuse

rules. Effective and automated rule mining techniques should be further explored, such as combining machine learning techniques [52] to characterize more context features, etc.

Details of API-misuse patterns. For clarity, we omit the class scope of APIs in Table III if no ambiguity will be caused.

`JSONObject.getString(String)=>JSONObject.optString(String)`. The former API will throw `JSONException` when the `JSONObject` does not has the query attribute (i.e., the given argument), which may crash the program if it is not handled. Therefore, to detect such misuses, we search the usages of the API where the exception is not properly tackled. Figure 6 shows 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, they are immediately confirmed and fixed after reporting to maintainers. `getJSONArray(String)` and `getJSONObject(String)` are similar.

`File.mkdir()=>File.mkdirs()`. Both `File.mkdir()` and `File.mkdirs()` are used to create a directory and return a boolean value to indicate whether the creation succeeds or not. The difference is the latter can recursively create the directories when nested paths do not exist, while `mkdir` cannot. The failure of directory creation may cause file access errors during program running, and is hard to debug. Therefore, when

```
    private Mpack downloadMpackMetadata(String
        mpackURI) throws IOException {
       File stagingDir = new File(mpackStaging.
           toString()+File.separator +
           MPACK_TAR_LOCATION);
       if (!stagingDir.exists()) {
---       stagingDir.mkdir();
+++       stagingDir.mkdirs();
       } ...} //https://github.com/apache/ambari/
           commit/b99bb28
```

Fig. 7: Accepted `File.mkdir()` misuse

```
    private void runBenchmarkTasks() throws
        Exception {
        ...
---     ExecutorService executor = Executors.
    newCachedThreadPool();
+++     ExecutorService executor = Executors.
    newFixedThreadPool(tasks.size());
        ...
    } //https://github.com/apache/bookkeeper/commit
        /0988e12
```

Fig. 8: Accepted `newCachedThreadPool()` misuse

creating a nested directory, the return value of the API call should be checked to avoid potential errors. Otherwise, `mkdirs` should be used to ensure the success of creation. Figure 7 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 `newFixedThreadPool(int)` can set the maximal thread number. 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 executors created from `newCachedThreadPool()` submit tasks in a loop without constraints. Figure 8 shows one accepted misuse in project *Apache bookkeeper*.

`String.replaceAll(String,String)=>String.replace(String,String)`. Both APIs replace all occurrences of

a `String` with others. However, the first argument for `replaceAll` is a regular expression, while plain text for `replace`. Compiling regex patterns will be more complex and consequently slower so we detect the misuse of the API `replaceAll` if it takes a plain `String` as the argument.

`Connection.createStatement()=>Connection.prepareStatement(String)`. Both APIs are used to execute SQL statements in Java. However, the former will highly degrade the performance of database access if intensively executing the same SQL statements in a loop. In this case, `prepareStatement(String)` should be used to enable the database to precompile the SQL statements and gain a better performance. We detect this kind of misuse by focusing on the case that an object of `Statement` created by `createStatement()` is used in a loop.

`Date.getTime()=>System.currentTimeMillis()`. We detect this kind of misuses by checking if the object of class `Date` only invokes the method `getTime()`. The reason is that `new Date()` for creating the `Date` object is simply a wrapper of method `System.currentTimeMillis()`. If it is intensively invoked in the program, the performance will be damaged. Using the method `System.currentTimeMillis()` can also avoid creating the temporary `Date` object.

`FileWriter.close()=>BufferedWriter.close()`. Indeed, this misuse is caused by creating a wrong writer object, i.e., `FileWriter` but not `BufferedWriter`. Large amount of input and output (IO) operations will significantly affect the performance of the program. `BufferedWriter` can effectively reduce the times of IO access with caches. Therefore, we detect such misuses by searching `FileWriter` object that is intensively used in a loop.

`String.equals(String)=>Objects.equals(String, String)`. Both APIs are used to check if two `String` values are same. However, It is possible that the first `String` in `String.equals(String)` may be NULL so that the `NullPointerException` would be thrown. We detect this kind of misuses by checking the possibility of causing `NullPointerException`. That is, there is no guard condition to check the nullness of the object before using.

As discussed above, a lot of submitted misuses have been accepted by maintainers. However, there are 15 misuses are rejected. We investigate these misuses and find some major reasons as follows. (1) There are no performance differences between two APIs in a sample or small project. For example, in project *Apache CXF*, the project maintainer rejected our submitted misuse by claiming that `System.currentTimeMillis()` and `new Date().getTime()` would not make the difference since this case occurs in a sample (small) project under *Apache CXF*. In *Apache NetBeans*, the project maintainer doubts that the change `File.mkdir()=>File.mkdirs()` is just a theoretical problem so they reject our pull request. (2) The submitted cases will change the code style of the entire project. For example, in project *Apache NetBeans*, the project maintainers claim that the change `Date.getTime()=>System.currentTimeMillis()` will reduce the readability since other `Date` cases can not be changed due to the context.

Finding 4: Based on the 10 Replaced misuse patterns, we have reported 149 misuses in latest Apache projects; 57 of them have been fixed by project maintainers so far.

V. Threats to Validity

The threats to external validity lie in the dataset used. To collect a large set of data for analysis, we mined bug-fixing commits from GitHub repositories. The dataset may be noisy for different reasons (i.e., not real bug fixes). We also define API misuses as code edit operations related to some APIs in a bug fix. In fact, it may also inaccurate results. The reasons are twofold. First, API misuses may occur in regular code changes, while we compute the percentage of API misuses over the operations from bug-fixing commits. Second, edit operations related to APIs may be not real API misuses.

The threats to internal validity relate to our implementation. To reduce errors, we use GumTree to extract AST operations, which is widely used in previous studies [30], [36], [37], [56]. However, we cannot get certain misuse categories, such as missing NULL checks directly from GumTree. Therefore, we revise GumTree by adding detailed program analysis to map operations to our classifications. To mitigate the threats of categorization noise from GumTree, we sample 100 operations for each category and confirm that 76% operations are correctly classified. Furthermore, we carefully review our code and scripts to ensure their correctness as much as we can.

VI. Conclusion

In this paper, we conduct an extensive empirical study on API misuses based on 528,546 bug-fixing commits. We extract fine-grained edit operations on AST of source code and classify them into different categories of API misuses. We also extract various frequent API-misuse patterns based on the categories. The results show that API misuses are prominent in practice and provide a set of guidelines for future research. Finally, based on our dataset, we perform a user study to manually analyze the usage constraints of 10 patterns to explore whether the mined patterns can guide the design of future API-misuse detection tools. The results show that 57 misuses (out of 149 reported misuses) have been fixed, indicating the importance of historical API misuses and the promising future for API-misuse detection. However, the current implementation still depends on our handcrafted detection rules. Effective and automated rule mining techniques should be further explored, such as combining machine learning [52] to characterize more context features, etc.

All experimental data and source code are open-source that can be downloaded at: https://github.com/BID3/BID3.

Acknowledgements

This work was partially supported by National Key Research and Development Program of China under Grant No. SQ2019YFE010068, National Science Foundation under Grant Nos. CCF-1763906 and CCF-1942430, Alibaba, and National Natural Science Foundation of China under No. 61922003.

REFERENCES

[1] 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.

[2] M. Monperrus and M. Mezini, "Detecting missing method calls as violations of the majority rule," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 22, no. 1, p. 7, 2013.

[3] "Github website," 2019. [Online]. Available: https://github.com/

[4] "Sourceforge website," 2019. [Online]. Available: https://sourceforge.net/

[5] "Jira website," 2019. [Online]. Available: https://www.atlassian.com/software/jira/

[6] M. Martinez and M. Monperrus, "Mining software repair models for reasoning on the search space of automated program fixing," *Empirical Softw. Engg.*, vol. 20, no. 1, pp. 176–205, Feb. 2015. [Online]. Available: http://dx.doi.org/10.1007/s10664-013-9282-8

[7] C. Lindig, "Mining patterns and violations using concept analysis," in *The Art and Science of Analyzing Software Data*. Elsevier, 2015, pp. 17–38.

[8] A. Wasylkowski, A. Zeller, and C. Lindig, "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, 2007, pp. 35–44.

[9] M. Acharya and T. Xie, "Mining api error-handling specifications from source code," in *International Conference on Fundamental Approaches to Software Engineering*. Springer, 2009, pp. 370–384.

[10] S. Thummalapenta and T. Xie, "Mining exception-handling rules as sequence association rules," in *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 2009, pp. 496–506.

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

[12] S. Amann, S. Nadi, H. A. Nguyen, T. N. Nguyen, and M. Mezini, "Mubench: a benchmark for api-misuse detectors," in *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*. IEEE, 2016, pp. 464–467.

[13] "Github archive website," 2019. [Online]. Available: https://www.gharchive.org/

[14] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. ACM, 2014, pp. 313–324.

[15] M. Wen, Y. Liu, R. Wu, X. Xie, S.-C. Cheung, and Z. Su, "Exposing library api misuses via mutation analysis," in *Proceedings of the 41st International Conference on Software Engineering*. IEEE Press, 2019, p. 866877. [Online]. Available: https://doi.org/10.1109/ICSE.2019.00093

[16] L. Zhang, T. Xie, L. Zhang, N. Tillmann, J. De Halleux, and H. Mei, "Test generation via dynamic symbolic execution for mutation testing," in *2010 IEEE International Conference on Software Maintenance*. IEEE, 2010, pp. 1–10.

[17] G. Liang, Q. Wang, T. Xie, and H. Mei, "Inferring project-specific bug patterns for detecting sibling bugs," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 2013, pp. 565–575.

[18] D. Hovemeyer and W. Pugh, "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, 2007, pp. 9–14.

[19] C. C. Williams and J. K. Hollingsworth, "Automatic mining of source code repositories to improve bug finding techniques," *IEEE Transactions on Software Engineering*, vol. 31, no. 6, pp. 466–480, 2005.

[20] X. Li, W. Li, Y. Zhang, and L. Zhang, "Deepfl: Integrating multiple fault diagnosis dimensions for deep fault localization," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 169–180.

[21] X. Li and L. Zhang, "Transforming programs and tests in tandem for fault localization," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, pp. 1–30, 2017.

[22] L. Zhang, L. Zhang, and S. Khurshid, "Injecting mechanical faults to localize developer faults for evolving software," in *OOPSLA*, 2013, pp. 765–784.

[23] L. Zhang, M. Kim, and S. Khurshid, "Localizing failure-inducing program edits based on spectrum information," in *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, 2011, pp. 23–32.

[24] M. Papadakis and Y. Le Traon, "Metallaxis-fl: mutation-based fault localization," *Software Testing, Verification and Reliability*, vol. 25, no. 5-7, pp. 605–628, 2015.

[25] S. Moon, Y. Kim, M. Kim, and S. Yoo, "Ask the mutants: Mutating faulty programs for fault localization," in *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, 2014, pp. 153–162.

[26] Y. Lou, A. Ghanbari, X. Li, L. Zhang, H. Zhang, D. Hao, and L. Zhang, "Can automated program repair refine fault localization? a unified debugging approach," in *ISSTA*, 2020, to appear.

[27] X. B. D. Le, D. Lo, and C. L. Goues, "History driven program repair," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, March 2016, pp. 213–224.

[28] A. Ghanbari, S. Benton, and L. Zhang, "Practical program repair via bytecode mutation," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 19–30.

[29] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *Ieee transactions on software engineering*, vol. 38, no. 1, pp. 54–72, 2011.

[30] J. Jiang, Y. Xiong, H. Zhang, Q. Gao, and X. Chen, "Shaping program repair space with existing patches and similar code," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2018, pp. 298–309.

[31] H. Zhong and Z. Su, "An empirical study on real bug fixes," in *Proceedings of the 37th International Conference on Software Engineering*, ser. ICSE '15, 2015, pp. 913–923. [Online]. Available: http://dl.acm.org/citation.cfm?id=2818754.2818864

[32] M. Soto, F. Thung, C. Wong, C. Le Goues, and D. Lo, "A deeper look into bug fixes: Patterns, replacements, deletions, and additions," in *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, May 2016, pp. 512–515.

[33] K. Pan, S. Kim, and E. J. Whitehead, Jr., "Toward an understanding of bug fix patterns," *Empirical Softw. Engg.*, vol. 14, no. 3, pp. 286–315, 2009. [Online]. Available: http://dx.doi.org/10.1007/s10664-008-9077-5

[34] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, 2014, pp. 437–440.

[35] V. Sobreira, T. Durieux, F. Madeiral, M. Monperrus, and M. de Almeida Maia, "Dissection of a bug dataset: Anatomy of 395 patches from defects4j," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2018, pp. 130–140.

[36] M. Martinez and M. Monperrus, "Coming: A tool for mining change pattern instances from git commits," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 2019, pp. 79–82.

[37] F. Madeiral, T. Durieux, V. Sobreira, and M. Maia, "Towards an automated approach for bug fix pattern detection," *arXiv preprint arXiv:1807.11286*, 2018.

[38] Z. Wan, D. Lo, X. Xia, and L. Cai, "Bug characteristics in blockchain systems: A large-scale empirical study," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, May 2017, pp. 413–424.

[39] N. Meng, S. Nagy, D. D. Yao, W. Zhuang, and G. A. Argoty, "Secure coding practices in java: Challenges and vulnerabilities," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: ACM, 2018, pp. 372–383. [Online]. Available: http://doi.acm.org/10.1145/3180155.3180201

[40] Q. Hanam, F. S. d. M. Brito, and A. Mesbah, "Discovering bug patterns in javascript," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: ACM, 2016, pp. 144–156. [Online]. Available: http://doi.acm.org/10.1145/2950290.2950308

[41] R. Yue, N. Meng, and Q. Wang, "A characterization study of repeated bug fixes," in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2017, pp. 422–432.

[42] H. A. Nguyen, A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, and H. Rajan, "A study of repetitiveness of code changes in software evolution," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Nov 2013, pp. 180–190.

[43] B. Ray, V. Hellendoorn, S. Godhane, Z. Tu, A. Bacchelli, and P. Devanbu, "On the "naturalness" of buggy code," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. ACM, 2016, pp. 428–439. [Online]. Available: http://doi.acm.org/10.1145/2884781.2884848

[44] B. Livshits and T. Zimmermann, "Dynamine: finding common error patterns by mining software revision histories," in *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5. ACM, 2005, pp. 296–305.

[45] Z. Li and Y. Zhou, "Pr-miner: automatically extracting implicit programming rules and detecting violations in large software code," in *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5. ACM, 2005, pp. 306–315.

[46] M. Monperrus, M. Bruch, and M. Mezini, "Detecting missing method calls in object-oriented software," in *European Conference on Object-Oriented Programming*. Springer, 2010, pp. 2–25.

[47] 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 *ESEC/FSE*. New York, NY, USA: ACM, 2009, pp. 383–392. [Online]. Available: http://doi.acm.org/10.1145/1595696.1595767

[48] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. Al-Kofahi, and T. N. Nguyen, "Recurring bug fixes in object-oriented programs," in *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering-Volume 1*. ACM, 2010, pp. 315–324.

[49] M. K. Ramanathan, A. Grama, and S. Jagannathan, "Path-sensitive inference of function precedence protocols," in *29th International Conference on Software Engineering (ICSE'07)*, May 2007, pp. 240–250.

[50] M. K. Ramanathan, A. Grama, and S. Jagannathan, "Static specification inference using predicate mining," in *ACM SIGPLAN Notices*, vol. 42, no. 6. ACM, 2007, pp. 123–134.

[61] "Infer website," 2019. [Online]. Available: https://fbinfer.com/

[51] A. Wasylkowski and A. Zeller, "Mining temporal specifications from object usage," in *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '09, 2009, pp. 295–306. [Online]. Available: https://doi.org/10.1109/ASE.2009.30

[52] T. T. Nguyen, H. V. Pham, P. M. Vu, and T. T. Nguyen, "Recommending api usages for mobile apps with hidden markov model," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 795–800.

[53] M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White, and D. Poshyvanyk, "An empirical investigation into learning bug-fixing patches in the wild via neural machine translation," in *ASE*, 2018, pp. 832–837.

[54] J. Jiang, L. Ren, Y. Xiong, and L. Zhang, "Inferring program transformations from singular examples via big code," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 255–266.

[55] A. Milanova, A. Rountev, and B. G. Ryder, "Parameterized object sensitivity for points-to analysis for java," *ACM Trans. Softw. Eng. Methodol.*, vol. 14, no. 1, p. 141, Jan. 2005.

[56] K. Liu, D. Kim, T. F. Bissyande, S. Yoo, and Y. Le Traon, "Mining fix patterns for findbugs violations," *IEEE Transactions on Software Engineering*, 2018.

[57] A. Hajnal and I. Forgacs, "A precise demand-driven definition-use chaining algorithm," in *Proceedings of the Sixth European Conference on Software Maintenance and Reengineering*, March 2002, pp. 77–86.

[58] M. J. Harrold and M. L. Soffa, "Efficient computation of interprocedural definition-use chains," *ACM Trans. Program. Lang. Syst.*, no. 2, pp. 175–204, 1994.

[59] S. Nielebock, R. Heumüller, and F. Ortmeier, "Commits as a basis for api misuse detection," in *Proceedings of the 7th International Workshop on Software Mining*, ser. SoftwareMining 2018. New York, NY, USA: ACM, 2018, pp. 20–23. [Online]. Available: http://doi.acm.org/10.1145/3242887.3242890

[60] "Findbugs website," 2019. [Online]. Available: http://findbugs.sourceforge.net/

[62] "Spotbugs website," 2019. [Online]. Available: https://spotbugs.github.io/