SCIENCE CHINA
Information Sciences

• RESEARCH PAPER •

March 2021, Vol. 64 132101:1–132101:19
https://doi.org/10.1007/s11432-019-2745-5

Identifying change patterns of API misuses from code changes

Wenjian LIU[1,2,3], Bihuan CHEN[1,2,3*], Xin PENG[1,2,3], Qinghao SUN[1,2,3] & Wenyun ZHAO[1,2,3]

[1]*School of Computer Science, Fudan University, Shanghai* 201203*, China;*
[2]*Shanghai Key Laboratory of Data Science, Fudan University, Shanghai* 201203*, China;*
[3]*Shanghai Institute of Intelligent Electronics & Systems, Shanghai* 200433*, China*

Received 23 September 2019/Revised 13 December 2019/Accepted 26 December 2019/Published online 7 February 2021

Abstract Library or framework APIs are difficult to learn and use, leading to unexpected software behaviors or bugs. Hence, various API mining techniques have been introduced to mine API usage patterns about the co-occurring of API calls or pre-conditions of API calls. However, they fail to mine patterns about an API call itself (e.g., whether the arguments of the API call are correctly set and whether the API is suitably chosen over other similar APIs). To bridge this gap, we propose CPAM to identify change patterns (in the form of a pair of APIs before and after code changes) to fix API misuses, using historical code changes. Given a set of target APIs and a corpus of open-source projects, CPAM first selects the commits that potentially fix API misuses from the corpus, then extracts changes to API misuses in each selected commit, and finally identifies change patterns of API misuses. We implement CPAM for Java, and conduct large-scale evaluation, targeting Java SE APIs and using a corpus of 1162 Java projects. Our experimental results demonstrate CPAM's effectiveness and efficiency. By applying identified change patterns to bug detection, we find 44 new bugs, and 18 of them have been confirmed and fixed.

Keywords API misuses, API mining, AST differencing, change patterns, bug fixing

Citation Liu W J, Chen B H, Peng X, et al. Identifying change patterns of API misuses from code changes. Sci China Inf Sci, 2021, 64(3): 132101, https://doi.org/10.1007/s11432-019-2745-5

1 Introduction

Libraries and frameworks can be reused through application programming interfaces (APIs) to improve development efficiency and quality of software systems. In practice, it is difficult, even for experienced developers, to learn and use APIs [1–4] owing to the fact that usage constraints of an API are often hidden assumptions [5], API combinations can be complicated [6], API documentation can often be incomplete or ambiguous [7], and APIs can be unstable [8,9] or break [10–13] as they evolve. As a result, developers may misuse APIs, which cause unexpected software behaviors or bugs (e.g., performance bugs [14–17] and security bugs [18–21]).

As open-source projects contain a wealth of knowledge about API usages, a variety of API usage mining techniques have been developed to extract API usage patterns. Such techniques can be mainly categorized into two types: snapshot-based (e.g., [6,22–28]) and history-based (e.g., [29–32]). The former mines from the source code snapshot of each project, while the latter mines from the newly-added code between historical revisions of each project. Thus, each extracted API usage pattern frequently occurs in code snapshots or added code.

Basically, a pattern, extracted by existing techniques, is in four forms. First, it can be a set of API calls that frequently co-occur [22–24, 32, 33]. For example, {FileInputStream.read, FileInputStream.close} describes that FileInputStream.read is called whenever FileInputStream.close is invoked, and vice versa. Second, it can be a sequence of API calls that frequently co-occur in a certain order [6,25–28]. For example,

* Corresponding author (email: bhchen@fudan.edu.cn)

FileInputStream.read → FileInputStream.close represents that FileInputStream.read cannot be called after FileInputStream.close is called. Third, it can be a guard condition of an API call, which has to be satisfied before the API is invoked [28,34–37]. For example, File.exists has to return false before File.createNewFile is called. Fourth, it can be a graph of API usages, which further takes into account the control and data flow among API usages [38,39]. In summary, these patterns can effectively capture co-occurring of API calls or pre-conditions of API calls, but are not designed to systematically characterize the single API call itself, e.g., whether the arguments are correctly set, whether the API is suitably chosen over other similar APIs, and whether the return value is checked against special values.

To fill this gap, we propose Cpam to identify change patterns to fix API misuses from revision history of open-source projects. Our key idea is that revision history can reflect API misuses via API usage changes that fix misuses, but project snapshots alone cannot. Moreover, only leveraging the added code as existing history-based techniques did is not sufficient as the deleted code gives hints on why an API is misused and how it is fixed. Thus, Cpam detects change patterns as a pair of APIs before and after changes, using fine-grained code change analysis that considers both the deleted and added code. To the best of our knowledge, this is the first study to systematically detect change patterns of a single API call itself, which serves as an effective complement to existing API usage mining techniques.

Cpam works in three steps. First, it uses several heuristics to select the commits that potentially contain API usage changes to fix API misuses, from a corpus of open-source projects. Second, it uses an abstract syntax tree (AST) differencing method [40] to generate fine-grained code changes for each selected commit, and identifies changes to API misuses using three change types systematically derived from potential changes to the composing elements of an API call (i.e., return value, receiver, API name and arguments). Finally, it uses in-project and cross-project frequencies to identify the change patterns of API misuses.

We implement Cpam for Java. To demonstrate Cpam's effectiveness, efficiency and potential applications, we conduct large-scale experiments, which target Java SE APIs and use a corpus of 1162 highly-starred Java projects on GitHub. We report the top ten change patterns of API misuses of the three change types, and manually analyze the changes of API misuses to evaluate the false positives. Results show that Cpam can effectively identify interesting change patterns of API misuses that can be useful for bug detection, with a false positive rate of 11.4% and acceptable performance.

In summary, this study makes the following contributions.

• We propose Cpam to identify change patterns to fix API misuses through fine-grained code change analysis.

• We conduct large-scale experiments on 1162 Java projects to evaluate Cpam's effectiveness, efficiency and applications.

• We discover 44 new bugs based on our detected change patterns; and 18 of them have been confirmed and fixed.

2 Preliminaries

In this section, we define our scope of API usage changes and introduce the AST differencing method we use in Cpam.

2.1 Scope of API usage changes

As we focus on API usages, we distinguish an API call from an API declaration. The only required elements of an API declaration are return type, API name, parameters, and API body, where the return type can be void and the parameters can be empty. The elements of an API call are return value, receiver, API name and arguments. Here, the return value does not exist if the API is declared as void, the receiver does not exist if the API is declared as static, and the arguments can be empty.

Our scope of API usage changes is defined as changes to the composing elements of an API call. We clarify that changes to the context of an API call, including API call sequences (e.g., adding/deleting an API call) and guard conditions (e.g., adding/deleting a conditional check to an API call or its arguments), are out of our scope. In other words, our study is complementary to existing API usage mining techniques that mainly focus on the mining of the context of an API call.

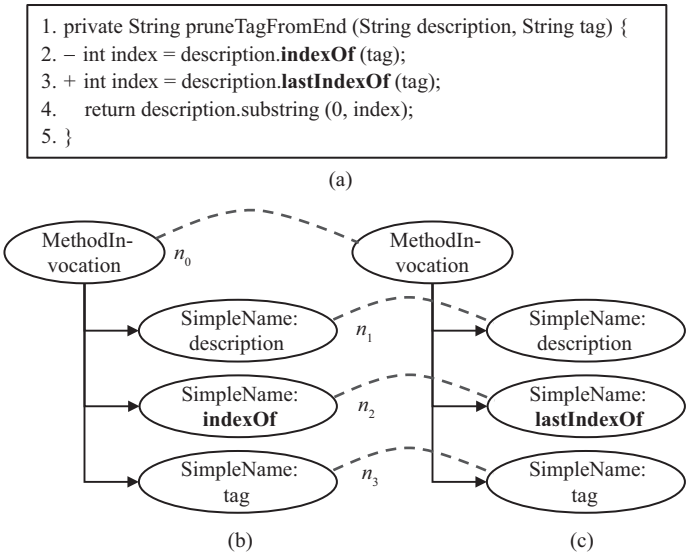

Figure 1 Replacing with an API call in the same class. (a) Code changes adapted from groovy-core; (b) partial AST before changes; (c) partial AST after changes.

2.2 AST differencing

An AST is a labeled ordered rooted tree whose nodes can contain string values. The label of a node represents a structural element of the source code, and the value indicates the actual token in the code. Specifically, the AST of an API call is organized in an ordered way with respect to its composing elements.

Example 1. Figure 1(b) presents the partial AST of the code in Figure 1(a). It corresponds to the changed API call at Line 2, and has four nodes n_0, n_1, n_2 and n_3. The label of n_0 is MethodInvocation, and the labels of $n1$, n_2 and n_3 are SimpleName. n_1 is the first child node of n_0, denoting the receiver of the API call; and its value is "description". The second child node n_2 denotes the API name, whose value is "indexOf". n_3 is the third one, indicating the argument of the API call; and its value is "tag".

AST differencing methods generate an edit script (i.e., a sequence of edit actions) between two ASTs. These edit actions can transform one AST into the other. Here, we use the state-of-the-art method GumTree [40]. It first uses heuristics to generate a mapping between the nodes in two ASTs in two successive phases. (1) It uses a greedy top-down algorithm to find isomorphic subtrees and establish mappings between the nodes of these isomorphic subtrees. (2) It uses a bottom-up algorithm to match two nodes if their descendants include a large number of anchors. Based on this mapping, it then generates the edit script in terms of four kinds of edit actions, i.e., update, add, delete, and move.

- update(n, v) replaces the value of node n with a new value v.
- add(n, p, i) inserts a new node n as the i-th child node of node p if p is not null. Otherwise, n becomes the new root node and has the previous root node as its only child.
- delete(n) removes a leaf node n.
- move(n, p, i) moves a node n to be the i-th child node of node p. All child nodes of n are moved together with n.

Example 2. Figure 1(c) shows the AST of the API call at Line 3 that replaces the API call at Line 2. GumTree generates the mapping as illustrated by the dotted lines between Figure 1(b) and (c). Based on the mapping, GumTree generates the edit script, which has one edit action update(n_2, lastIndexOf), for the code change in Figure 1(a). This edit action updates the API name from "indexOf" to "lastIndexOf".

3 Methodology

Figure 2 presents an overview of the proposed approach Cpam. The inputs of Cpam are a set of target APIs and a corpus of open-source projects with their revision history. Cpam works in three steps: selecting commits that potentially contain API usage changes to fix API misuses (Subsection 3.1), extracting changes to API misuses via fine-grained code changes analysis (Subsection 3.2), and identifying change patterns of API misuses (Subsection 3.3). The change patterns are reported to users, which can be

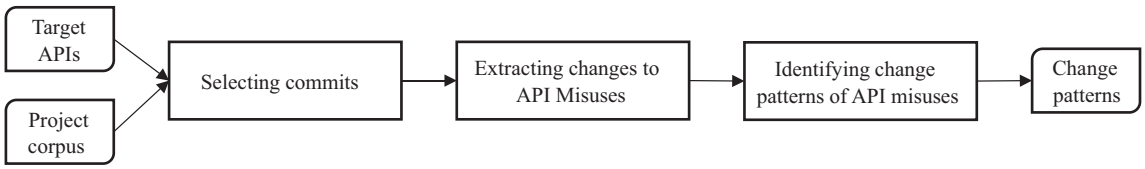

Figure 2 Overview of our approach.

analyzed to facilitate bug detection (i.e., enrich bug patterns of bug detection tools and detect new similar bugs in open-source projects). Here, we focus on APIs implemented in Java, though CPAM is actually general for other programming languages.

3.1 Selecting commits

As we target API misuses, the first step of CPAM is to select the commits that potentially contain API usage changes that fix API misuses, from the revision history (i.e., a list of commits) of the corpus. For this purpose, we design three heuristics.

• H1. Identifying bug-fix commits. As API misuses often lead to unexpected behaviors (or logic bugs) or crashes (or bugs), we identify bug-fix commits by searching for related keywords in commit logs, which has been commonly used in the literature. In detail, we use four sets of keywords from the literature to locate the commits that fix bugs [41] (e.g., "bug", "patch", and "fix"), performance bugs [42, 43] (e.g., "performance", "slow", and "throughput"), security bugs [44] (e.g., "security" and "insecure"), and compatibility bugs [45] (e.g., "compatibility" and "compatible"). These four sets of keywords are chosen to cover a broad range of software bugs API misuse can cause.

• H2. Filtering commits with large code changes. The commits with large code changes can contain many API usage changes that are not related to commit logs, as developers may tangle unrelated changes into one commit [46, 47]. Hence, the bug-fix commits identified by H1 can have API usage changes that are not related to bugs causing false positives of API misuses. On the other hand, the commits with small code changes are more relevant to corrective activities [48], which can exclude falsely-identified bug-fix commits in H1. Thus, we select bug-fix commits with small code changes; i.e., we filter the commits that change more than 5 source files or 50 lines of code in a file, where the thresholds are set based on [48].

• H3. Filtering commits with only added or deleted code. Only adding code but not deleting code in a commit, or only deleting code but not adding code does not involve any changes to the composing elements of API calls, but only adds or deletes the whole API calls or API guard conditions. These changes are out of our scope of API usage changes, as introduced in Subsection 2.1. Thus, we filter this kind of commits.

Specifically, we first apply H1 on the revision history of each project, and then apply H2 and H3 on the bug-fix commits identified by H1 to perform the further filtering.

3.2 Extracting changes to API misuses

Given the selected commits of each project in the corpus, the second step of CPAM is to extract changes to fix API misuses by analyzing fine-grained code changes. Before diving into the details, we first systematically derive five change types from any individual or combined change to the four composing elements (as introduced in Subsection 2.1) of an API call. Note that the receiver and arguments of an API call are objects with a specific type.

• T1. Modifying arguments of an API call. Developers may pass wrong or incorrect arguments to call an API [49] or pass arguments in the wrong order when calling an API with multiple parameters of the same type [50–52]. Thus, an individual change to an API call's arguments (i.e., passing different arguments of the same type) can reflect such an API misuse.

• T2. Replacing an API call with an API call in the same class. A class can contain multiple functionally-similar APIs. Such APIs can have different API names (with the same or different parameters), or have the same API name but different parameters (i.e., method overloading). Developers might choose a wrong or less suitable API in a class. Hence, a replacement of two API calls, i.e., an individual change to the API name or a combined change to the API name, and arguments (i.e., using a different API), and an individual change to the arguments (i.e., using an overloaded API), can capture the API misuse.

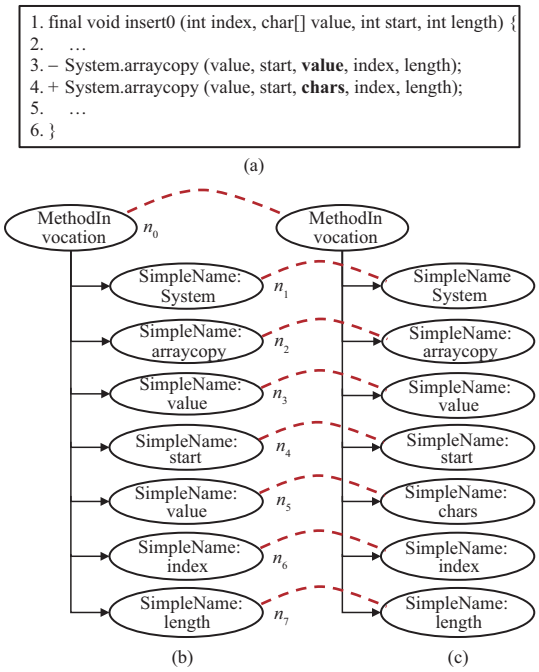

Figure 3 (Color online) Modifying arguments of an API call. (a) Code changes adapted from libgdx; (b) partial AST before changes; (c) partial AST after changes.

• T3. Replacing an API call with an API call from a different class. Different APIs from different classes can have similar functionalities. Similar to T2, developers may use a wrong or less suitable API, which could be captured by an individual change to the receiver of an API call or a combined change to the receiver, API name and arguments (i.e., using a different receiver that has a different type).

• T4. Adding checks to the return value of an API call. An API may return special values to indicate whether the API call is successful or not or to indicate the error code of the API call. However, developers may forget to check the return value of an API call against special values, making the program stay in a potentially error state and finally causing a bug. Thus, added checks to return values can reflect such an API misuse. As T4 has been studied [53], we omit this type from our approach.

• T5. Changing the receiver of an API call. It is also possible to use a different receiver of the same type to call an API; i.e., developers call an API with a new receiver. Such receiver changes are often not related to APIs (as will be evaluated in Subsection 4.2.5). Hence, we omit this type from our approach.

Example 3. Figure 3(a) illustrates an example of T1. Variables value and chars have the same type. The wrong variable value is passed to the third argument of arraycopy. As a result, the content in value is copied to value itself, but not to chars. Figure 1(a) shows an example of T2, where indexOf is used to prune the tag from the front. As indicated by the method name at Line 1, lastIndexOf, from the same class of indexOf, should be used to prune the tag from the end. Figure 4(a) reports an example of T3. System.arraycopy is favored over Arrays.copyOf for better performance.

Then, we introduce how to extract changes to API misuses of types T1–T3. For each selected commit, we first apply GumTree [40] on each changed Java source code file to generate fine-grained code changes; i.e., a list of edit actions (as introduced in Subsection 2.2). We then iterate each edit action $\mathcal{A}$ to extract the API usage change and determine its type.

If $\mathcal{A}$ is update(n, v), we check whether the label of n's parent node is MethodInvocation and whether this method invocation calls a target API by resolving type bindings with JDT parser. If not, we skip $\mathcal{A}$; and if yes, $\mathcal{A}$ is related to a target API call, and we check n's position in the subtree of this method invocation.

• If n is the first child node, the receiver is changed. If the receiver's type is not changed (i.e., using a different receiver of the same type), we skip $\mathcal{A}$ because this change is T5. If the receiver's type is changed, we classify this change as T3, and record it as a 3-tuple $\langle \text{api}_\text{b}, \text{api}_\text{a}, \text{T3} \rangle$, where api_b and api_a denote the API signature before and after the change. Hereafter, api_b and api_a have the same meaning.

• If n is the second child node, the API name is changed. If the first child node of this method

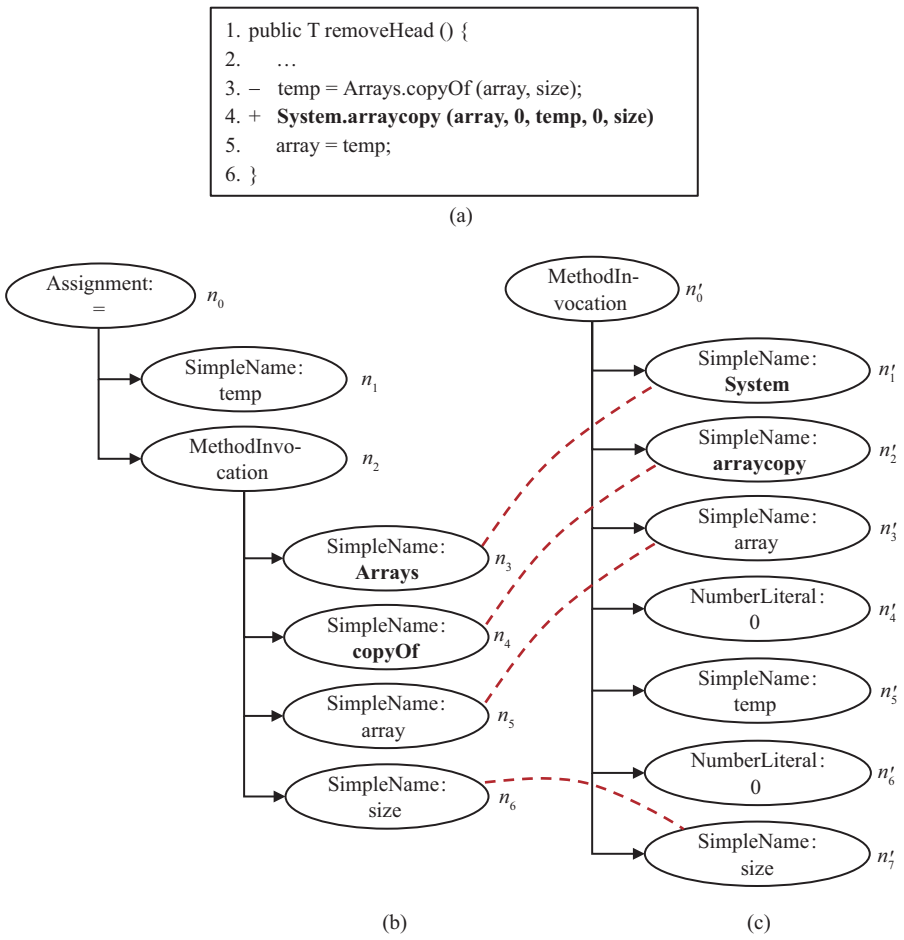

Figure 4 (Color online) Replacing with an API call from a different class. (a) Code changes adapted from java-algorithms-implementation; (b) partial AST before changes; (c) partial AST after changes.

invocation is not changed after checking the mapping produced by GumTree (i.e., the receiver is not changed), we record this change as $\langle \text{api}_\text{b}, \text{api}_\text{a}, \text{T2}\rangle$; otherwise, we skip $\mathcal{A}$ because it is an associated change that results from the change to the first child node.

• If n is other child node, an argument is changed. If the first or second child node of this method invocation is changed, we skip $\mathcal{A}$ because it is the associated change; otherwise, we record this change as $\langle \text{api}_\text{b}, \text{api}_\text{a}, \text{T2}\rangle$ if the type of the argument is changed (i.e., this API call is replaced by an overloaded API call with the same number of arguments but a different type), or as $\langle \text{api}_\text{b}, l, \text{T1}\rangle$ (where l is the position of the changed argument) if the type of the argument is not changed.

If $\mathcal{A}$ is add(n, p, i) or delete(n), we check whether n denotes a method invocation to a target API. If yes, analyzing edit actions on n's child nodes can capture this change, and hence we skip $\mathcal{A}$; if not, we check whether n's parent node p corresponds to a method invocation to a target API. If not, we skip $\mathcal{A}$; and if yes, we check n's position in the subtree of this method invocation.

• If n is the first or second child node, there are two cases. First, an API call is purely added or deleted, which is out of our scope (as introduced in Subsection 2.1). Second, one API call is added and another API call is deleted, without being mapped by GumTree, which could occasionally happen owing to the imprecision in GumTree. Therefore, we skip $\mathcal{A}$.

• If n is other child node, an argument is inserted or removed. If the first or second child node of this method invocation is changed, we skip $\mathcal{A}$ because it is the associated change; otherwise, we record this change as $\langle \text{api}_\text{b}, \text{api}_\text{a}, \text{T2}\rangle$ (i.e., this API call is replaced by an overloaded API call with a different number of arguments).

If $\mathcal{A}$ is move(n, p, i), then if n denotes a method invocation to a target API, we skip $\mathcal{A}$ because a whole API call is moved but not changed. If n's parent node p corresponds to a method invocation to a target API, p must be newly added and all its other child nodes must be generated through other update, move

add(n'_0, *, *)	Skip (checking changes on its child nodes)
update(n_3, System)	T3
move(n_3, n'_0, 0)	Skip (not reflecting any change)
update(n_4, arraycopy)	Skip (associated change)
move(n_4, n'_0, 1)	Skip (not reflecting any change)
move(n_5, n'_0, 2)	Skip (not reflecting any change)
add(n'_4, n'_0, 3)	Skip (associated change)
add(n'_5, n'_0, 4)	Skip (associated change)
add(n'_6, n'_0, 5)	Skip (associated change)
move(n_6, n'_0, 6)	Skip (not reflecting any change)
delete(n_1)	Skip (not related to API call)
delete(n_2)	Skip (checking changes on its child nodes)
delete(n_0)	Skip (not related to API call)

Figure 5 Edit actions for Figure 4.

and add actions according to GumTree. As such move actions do not reflect any change to an API call, analyzing other update and add actions can capture the API usage change and hence such move actions are skipped.

Notice that a call to a static method usually does not have a receiver, but has the class name as the first child node in its AST. Thus, static method invocation does not break the previous procedure. An API call truly lacks a receiver when the whole qualified name of the API is statically imported, where the previous procedure breaks. To support this case, we check static imports and tune the order of child nodes in the previous procedure.

Example 4. The generated edit action for the two ASTs in Figures 1(b) and (c) is update(n_2, lastIndexOf), and this change is correctly classified as T2. Similarly, the edit action for the two ASTs in Figures 3(b) and (c) is update(n_5, chars), and this change is correctly classified as T1. The edit actions for the two ASTs in Figures 4(b) and (c) are reported in Figure 5 together with the reasons why an edit action is skipped. Only update(n_3, System) is not skipped, and this change is correctly classified as T3. Notice that * indicates omitted details as we give partial ASTs in Figure 4.

3.3 Identifying change patterns of API misuses

Given the extracted changes to API misuses, the third step of Cpam is to identify the change patterns via a ranking based on in-project frequency f_{in} and cross-project frequency f_{cr}. f_{in} and f_{cr} characterize the commonality of a change pattern from two different perspectives. f_{in} reflects the number of times that a change pattern occurs in all the projects, and f_{cr} captures the number of projects where a change pattern occurs. They are both important indicators of the significance of a change pattern.

We first present how to compute f_{in} and f_{cr} for a change pattern. If the change type is T1, the change pattern is represented as $\langle \text{api}_\text{b}, L\rangle$, and its f_{in} is computed as the number of extracted changes that belong to T1, share the same api_b and change the same set of arguments L (which is feasible since we record the position l of each changed argument). Thus we can provide fine-grained change patterns to the arguments of an API call. If the change type is T2 or T3, the change pattern is represented as $\langle \text{api}_\text{b}, \text{api}_\text{a}\rangle$, and its f_{in} is computed as the number of extracted changes that belong to T2 or T3 and share the same api_b and api_a. Thus, we can distinguish the replacements of an API call with a call to different APIs either in the same class or from a different class. Based on these extracted changes for computing f_{in}, f_{cr} is computed by counting the number of projects where these extracted changes come from.

Then, we introduce how to rank change patterns. Overall, we compute a weighted sum of f_{in} and f_{cr} for each change pattern (see (1)). The higher the score, the higher the rank.

$$\text{score} = w \times \frac{f_{\text{in}} - f_{\text{in}}^{\min}}{f_{\text{in}}^{\max} - f_{\text{in}}^{\min}} + (1-w) \times \frac{f_{\text{cr}} - f_{\text{cr}}^{\min}}{f_{\text{cr}}^{\max} - f_{\text{cr}}^{\min}}. \tag{1}$$

To allow a unified measurement of f_{in} and f_{cr} independent of their units and ranges, we normalize f_{in} (respectively f_{cr}) into a value between 0 and 1 via comparing it with the maximum value $f_{\text{in}}^{\max}$ (respectively $f_{\text{cr}}^{\max}$) and minimum value $f_{\text{in}}^{\min}$ (respectively $f_{\text{cr}}^{\min}$) of all change patterns. Here w is the weight that trades off f_{in} and f_{cr}, and we set w to 0.5 to give equal priority to f_{in} and f_{cr}. Notice that a

Table 1 Distribution of change types

Change type	Extracted changes (#)	Covered projects (#)
T1	4793 (288)	386
T2	3689 (271)	387
T3	4026 (117)	380
T4	4120	380
T5	4796	403
Missed	2119	280

larger value to w can help Cpam to detect more project-specific change patterns of API misuses, and a smaller value to w can help Cpam to detect more cross-project change patterns of API misuses.

4 Evaluation

We have implemented Cpam with around 6.0 K lines of Java code. We have made Cpam open-source at our website[1)], together with all the experimental data for the ease of replication.

4.1 Evaluation setup

We conduct large-scale experiments on 1162 open-source Java projects to evaluate the effectiveness, efficiency and applications of Cpam. We select these projects from GitHub based on their popularity (i.e., the number of stars is more than 1000) as popular projects are widely used and are expected to contain mature code and representative change patterns to mine from. In total, these projects have around 60 million non-blank and non-comment lines of code and 3 million commits. In the evaluation, we target Java SE APIs, but treat printing and logging APIs as uninterested as their usage changes are usually for debugging, maintenance or format updates. We conduct experiments, on an Intel E5-2403 1.8 GHz server with 8 GB RAM, to answer the following three research questions.

- RQ1. What are the detected change patterns of API misuses?
- RQ2. What is the performance overhead of Cpam?
- RQ3. Can Cpam be used to facilitate bug detection?

We manually analyze the false positives of the extracted changes to API misuses (i.e., the change of an API usage is not related to bug-fixing), which is required to fairly answer RQ1. To this end, three of the authors and two hired developers are involved in this manual analysis; and all of them have more than five years of Java programming experience. They are first randomly assigned a set of extracted changes to API misuses with their commit links. Then, they are asked to decide whether an API usage change fix an API misuse by analyzing the commit or relying on any useful materials (e.g., discussion in a linked issue). Moreover, they are asked to record uncertain cases and categorize the main sources of false positives. After completing the analysis, they participate in a group discussion to analyze uncertain cases together and summarize the main sources of false positives. After reaching a consensus about the sources of false positives, they review and relabel the source of false positives for each assigned API usage change.

4.2 Detected change patterns (RQ1)

4.2.1 *Change type distribution*

We omit change types T4 and T5 from our approach as T4 has been studied in [53] and T5 is usually not related to API misuses (see Subsection 3.2). To evaluate the completeness of our change type classification in Subsection 3.2, we implement the detection of T4 by following [53] and the detection of T5 via a straightforward extension to our approach in Subsection 3.2. If an extracted API usage change does not belong to T1–T5, it is considered as missed owing to our classification incompleteness. Table 1 reports the distribution of all extracted API usage changes against T1–T5 and missed changes. The second column reports the number of extracted changes that belong to a certain change type (where the number of change patterns occurring in at least two projects is reported in parentheses). The third column reports the number of projects covered by the extracted changes.

1) https://cpam2019.wixsite.com/mysite.

Table 2 Top ten change patterns of T1

Change pattern	In-Pro. Freq. (#)	Cross-Pro. Freq. (#)	False positives (#)			Supported by tools?		
			Mapping	Refactoring	Tangling	S.B.	E.P.	PMD
⟨Thread.sleep(long), {1}⟩	301	81	6	2	18	✗	✗	✗
⟨String.equals(Object), {1}⟩	139	42	8	13	2	✗	✗	✗
⟨StringBuilder.append(String), {1}⟩	63	33	5	14	0	✗	✗	✗
⟨String.format(String, Object...), {2}⟩	55	22	0	7	1	✗	✓	✗
⟨CountDownLatch.await(long, TimeUnit), {1}⟩	74	15	0	0	12	✗	✓	✗
⟨String.format(String, Object...), {3}⟩	31	18	0	5	1	✗	✓	✗
⟨String.substring(int, int), {2}⟩	23	19	2	4	1	✗	✗	✗
⟨System.arraycopy(Object, int, Object, int, int), {3}⟩	20	15	0	0	2	✗	✗	✗
⟨String.startsWith(String), {1}⟩	19	15	1	4	0	✗	✗	✗
⟨Math.max(int, int), {2}⟩	26	12	0	0	1	✗	✗	✗
Sum	751	–	22 (2.9%)	49 (6.5%)	38 (5.1%)	0	3	0

We can observe that T1–T4 are all quite common across more than 30% of projects; and we only miss 2119 (i.e., 9.0%) API usage changes. This indicates that our derived change types have a reasonable coverage of API usage changes, and thus are representative. Besides, T5 is common but not considered, whose rationality will be evaluated in Subsection 4.2.5.

4.2.2 *Change patterns of T1*

We analyze the top ten change patterns of T1. The results are reported in Table 2. The first column lists the change pattern, and the second and third columns list the in-project frequency (In-Pro. Freq.) and cross-project frequency (Cross-Pro. Freq.).

False positives. The fourth to sixth columns of Table 2 present statistics about false positives, which are derived via our manual analysis. Incorrect mapping in GumTree leads to a false positive rate of 2.9%, where a misused API call is falsely mapped to an irrelevant API call. This is particularly the case for commits with large, structural or complex code changes. This is one of the reasons that we pick commits with small code changes in the first step of Cpam (i.e., heuristic H2 in Subsection 3.1). Refactoring (e.g., renaming local variables) caused a false positive rate of 6.5%. Because refactoring is often tangled into bug-fix commits [46, 47], it can be falsely identified as an argument change of a misused API. Besides, we classify other tangled code changes that are not related to the commit message in bug-fix commits as false positives since we fail to know the intention of such code changes. Overall, this causes a false positive rate of 5.1%.

Case studies. The most common pattern is to change the argument of `sleep`, which happens 301 times in 81 projects. Of the true positives, 57% increase the argument value, mostly for avoiding too frequent requests of heavyweight events in order to prevent performance degradations or even denial-of-services. In such cases, `sleep` is usually called in a loop, manifesting poor performance. Besides, some decrease the argument value to avoid wasting time. This indicates that developers should carefully set the argument of `sleep` to avoid performance issues. Similar cases are also found for `CountDownLatch.await`.

String APIs `equals`, `append`, `format`, `substring` and `startsWith` are often involved in logic bugs, mostly caused by logic changes of string formats. This suggests that developers should clearly document the assumed string formats when using string APIs, and timely change their usage when the assumption breaks. Specifically, a commonly-seen pattern for `format` is to change the argument passed for formatting. This is often owing to the incompatibility between the passed argument and the string format placeholder specified in the first argument of `format`.

Figure 3 illustrates an interesting pattern about `arraycopy`, where the same variable is passed to both the source (the first argument) and the target (the third argument) array. Under such a circumstance, `arraycopy` might be buggy, especially when the second and the fourth arguments have the same value (which means that the original content in the array is copied to itself).

4.2.3 *Change patterns of T2*

We analyze the top ten change patterns of T2. The results are reported in Table 3.

False positives. Similar to T1, incorrect mapping, refactoring and tangling are the three sources of false positives. Differently, refactoring causes a much lower false positive because refactoring mostly affects the arguments of an API call.

Case studies. The most common pattern is to replace `equals` with `equalsIgnoreCase`, which happens 50 times in 28 projects. We find from these cases that `equalsIgnoreCase` is often favored in

Table 3 Top ten change patterns of T2

Change pattern	In-Pro. Freq. (#)	Cross-Pro. Freq. (#)	False positives (#)			Supported by tools?		
			Mapping	Refactoring	Tangling	S.B.	E.P.	PMD
⟨String.equals(Object), String.equalsIgnoreCase(String)⟩	50	28	0	0	0	✗	✗	✗
⟨String.isEmpty(), String.length()⟩	40	26	0	0	1	✗	✗	✗
⟨String.indexOf(String), String.contains(CharSequence)⟩	53	18	1	0	5	✗	✗	✗
⟨System.currentTimeMillis(), System.nanoTime()⟩	49	15	0	0	2	✗	✗	✗
⟨Integer.valueOf(String), Integer.parseInt(String)⟩	39	13	0	0	0	✓	✗	✗
⟨System.nanoTime(), System.currentTimeMillis()⟩	36	10	0	0	2	✗	✗	✗
⟨String.equals(Object), String.startsWith(String)⟩	22	16	4	0	1	✗	✗	✗
⟨Math.max(int, int), Math.min(int, int)⟩	24	12	7	1	0	✗	✗	✗
⟨String.length(), String.isEmpty()⟩	27	10	0	0	3	✗	✗	✗
⟨String.startsWith(String), String.equals(Object)⟩	18	12	1	0	0	✗	✗	✗
Sum	358	–	13 (3.6%)	1 (0.3%)	14 (3.9%)	1	0	0

```
1. private void copySelectedTaskNames() {
2.     String names = getSelectedTaskNames();
3. -   if( names.isEmpty() ) {
4. +   if (names.length() == 0) {
5.         ...
6. }
```

(a)

```
1. public void setUp() throws Exception {
2. -    assertTrue(message.indexOf(TEST_MESSAGE) >= 0);
3. +    assertTrue(message.contains(TEST_MESSAGE));
4.      ...
5. }
```

(b)

```
1. public long getDelay(TimeUnit unit) {
2. -    return unit.convert(endMs - System.currentTimeMillis(), TimeUnit.MILLISECONDS);
3. +    return unit.convert(endNs - System.nanoTime(), TimeUnit.NANOSECONDS);
4. }
```

(c)

```
1. protected void doRun() {
2.      ... // strategies need to wait at least one minute
3. -    final long waitInMins = Math.min(1, Math.max(60, c.getRetentionStrategy().check(c)));
4. +    final long waitInMins = Math.max(1, Math.min(60, c.getRetentionStrategy().check(c)));
5.      ...
6. }
```

(d)

Figure 6 Some change patterns of T2. Code changes taken from (a) gradle; (b) commons-lang; (c) exhibitor; (d) jenkins.

request and response processing in network-related projects, which can give some hints on detecting case-sensitivity logic bugs. Similarly, switches between `equals` and `startsWith` are logic bugs caused by business logic changes.

Figure 6(a) illustrates another common pattern, where the call to `isEmpty` is replaced with a call to `length` and a checking of its return value. This is owing to a compatibility issue: `isEmpty` is not available on older versions of JDK (e.g., 1.5) which are still widely used. Correspondingly, many projects do not use an older JDK version, and hence it is safe to replace `length` with `isEmpty` to improve code readability.

Figure 6(b) shows a common pattern that occurs 53 times in 18 projects, where indexOf is replaced with `contains` for improving code readability. In fact, `contains` is implemented by calling indexOf. Thus, the improved code readability is at a price of a slight performance degradation for an extra call.

Two interesting common patterns are about replacements between `currentTimeMillis` and `nanoTimes`. The former uses wall-clock time and may suffer leap seconds or large NTP corrections, while the latter is introduced in JDK 1.5 and uses a monotonic clock. As shown in Figure 6(c), `nanoTimes` should be used in accurate measurement of delayed or elapsed time; otherwise, bugs might occur. `currentTimeMillis` should be used in clock time computation.

Table 4 Top ten change patterns of T3

Change pattern	In-Proj. Freq. (#)	Cross-Proj. Freq. (#)	False positives (#)	Supported by tools? S.B.	E.P.	PMD
⟨Integer.parseInt(String), Long.parseLong(String)⟩	26	12	0	✗	✗	✗
⟨Arrays.asList(Object[]), Collections.singletonList(Object)⟩	21	13	3	✗	✓	✗
⟨Thread.sleep(long), CountDownLatch.await(long, TimeUnit)⟩	15	8	1	✗	✗	✗
⟨Arrays.asList(String[]), Collections.singletonList(String)⟩	14	8	0	✗	✓	✗
⟨String.equals(Object), Objects.equals(Object, Object)⟩	16	5	0	✗	✗	✗
⟨StringBuffer.append(char), StringBuilder.append(char)⟩	21	1	0	✗	✗	✗
⟨StringBuffer.append(String), StringBuilder.append(String)⟩	11	5	3	✗	✗	✗
⟨StringBuffer.toString(), StringBuilder.toString()⟩	10	5	0	✗	✗	✗
⟨Long.parseLong(String), Double.parseDouble(String)⟩	13	3	0	✗	✗	✗
⟨Integer.valueOf(String), Long.valueOf(String)⟩	6	6	0	✗	✗	✗
Sum	153	–	7 (4.6%)	0	2	0

Another common pattern is about numeric conversion, where `Integer.valueOf` is changed to `Integer.parseInt` to avoid boxing and unboxing to get an `int` value from a string. It improves the performance of creating a primitive from a string.

Figure 6(d) reports a pattern to change `min` to `max`, where the comment at Line 2 actually shows the evidence of such a logic bug. This is similar to Figure 1(a) where the method name reveals the misuse of indexOf at Line 2. It suggests that the semantic gap between an API call and its surrounding code elements can be used to detect such logic bugs. Notice that Rice et al. [52] recently leveraged the semantic gap between parameter names and argument names to detect many argument selection bugs.

4.2.4 *Change patterns of T3*

We analyze the top ten change patterns of T3. The results are reported in Table 4.

False positives. Different from T1 and T2, the false positives of T3 are all caused by tangled code changes.

Case studies. Three of the ten common patterns are about the range or incompatible type issues when a string is converted to a number. Similar to the string APIs in T1 (see Subsection 4.2.2), they are often related to the hidden or undocumented assumption of the converted string. However, the context of an API call could give some hints on such assumptions, and help detect such bugs. Figure 7(a) shows such an example: the return type of the method at Line 1 indicates that `Long.parseLong` should be used.

Figure 7(b) illustrates the switch from `Arrays.asList` to `Collections.singletonList` when only one element is stored. The difference between them is that the return value of `Collections.singletonList` is an immutable list that contains only the specified object, whereas the return value of `Arrays.asList` is a fixed-size list backed by an array. As a result, `Arrays.asList` might have a potential problem that the returned list may be changed accidentally.

Figure 7(c) describes a pattern that replaces `Thread.sleep` with `CountDownLatch.await`. When `Thread.sleep` is used to wait for something to finish, it has to sleep for the given waiting time even if something is already finished. Under such circumstances, `CountDownLatch.await` is more flexibility as it waits until the given waiting time elapses, or the latch has counted down to zero by calling `countDown`. Therefore, the performance can be sped up, especially when they are called in a loop and for the purpose of synchronization .

Another common pattern is to replace `String.equals` by `Object.equals` to prevent a null-point exception. As shown in Figure 7(d), `packageName` can be `null`, and hence it is not safe to call `String.equals` on `packageName`, but is safe to use `Object.equals`.

`StringBuffer.append` is thread-safe but has worse performance than `StringBuilder.append` that is not thread-safe. Therefore, a commonly-seen pattern is to replace the former with the latter in single-thread applications for improving performance.

4.2.5 *Evaluation of T5*

As reported in Table 1, we exclude a total of 4796 identified API usage changes of T5. To evaluate whether they are related to API misuses, we randomly sample and analyze 100 API usage changes. We find only one case that is related to API misuses, and most of them are caused by business logic changes, tangled code changes, or refactoring of the receiver object. This indicates that it is reasonable to exclude T5 from CPAM as CPAM is not designed to detect all API misuses but to identify API misuses with low false positives. The found case is interesting, as shown in Figure 8, where multiple consecutive calls to

```
1. public long getBodySize() {
2.     ...
3. -   return Integer.parseInt(this.headers.get("content-length"));
4. +   return Long.parseLong(this.headers.get("content-length"));
5. }
```

(a)

```
1. -  private List<String> allowedAnnotations = Arrays.asList("Override");
2. + private List<String> allowedAnnotations = Collections.singletonList("Override");
```

(b)

```
 1. public void testTtl_issue1783() throws IOException, InterruptedException {
 2.       ...
 3. +     final CountDownLatch latch = new CountDownLatch(1);
 4. +     mapConfig.addEntryListenerConfig(new EntryListenerConfig()
 5. +           .setImplementation(new EntryAdapter() {
 6. +                 @Override
 7. +                 public void entryEvicted(EntryEvent event) {
 8. +                       latch.countDown();
 9. +                 }
10. +        }));
11.       ...
12. -     Thread.sleep(4000);
13. +     assertTrue(latch.await(30, TimeUnit.SECONDS));
14.       ...
15. }
```

(c)

```
1. public boolean equals(Object otherObj) {
2.     ...
3. -   if (!packageName.equals(other.packageName)) {
4. +   if (!Objects.equals(packageName, other.packageName)) {
5.         return false;
6. }
```

(d)

Figure 7 Some change patterns of T3. Code changes taken (a) nanohttpd, (b) checkstyle, (c) hazelcast, (d) platform_frameworks_base.

`append` are chained together for improved performance. The reason is that Java virtual machine (JVM) has a special optimization for string concatenation and it recognizes the pattern in chained calls but not in consecutive calls.

CPAM successfully detect common change patterns to fix API misuses with respect to the three change types. These patterns either suggest bug detection patterns or good programming practices. On average, CPAM has a false positive rate of 11.4% owing to the incorrect mapping in GUMTREE, refactoring and tangled code changes.

4.3 Performance overhead (RQ2)

The performance of CPAM is closely related to the number of commits that are selected from the revision history for further analysis. Hence, we report the statistics about commits in the second to fourth columns of Table 5. In total, there are about 3 million commits in all the projects; after the filtering by H1, 0.7 million bug-fix commits remain; and further filtered by H2 and H3, 0.3 million commits are finally selected. On average, 276 commits are selected from each project.

The last four columns of Table 5 report the statistics about the time overhead of each step in CPAM, i.e., selecting commits (Step 1), AST differencing (Step 2), and identifying changes to fix API misuses and detecting change patterns of API misuses (Step 3). On average, for each project, it takes 0.8 s to select commits, 222.0 s to conduct AST differencing, and 40.0 s to identify changes and detect patterns. About 4.4 min are needed for CPAM to analyze one project, and 3.7 min are consumed by AST differencing. In total, it takes about 85 h to analyze 1162 projects, which is acceptable.

The overall performance of CPAM is acceptable, taking 4.4 min for each project. The most expensive step is the application of AST differencing to generate edit actions.

```
1. public String toString() {
2.     ...
3. -    StringBuilder buf = new StringBuilder();
4. -    buf.append(StringUtil.simpleClassName(this));
5. -    buf.append("(ridx: ");
6. -    buf.append(readerIndex);
7. -    buf.append(", widx: ");
8. -    buf.append(writerIndex);
9. -    buf.append(", cap: ");
10. -   buf.append(capacity());
11. +   StringBuilder buf = new StringBuilder().append(StringUtil.simpleClassName(this))
12. +   .append("(ridx: ").append(readerIndex).append(", widx: ").append(writerIndex)
13. +   .append(", cap: ").append(capacity());
14.     ...
15. }
```

Code Changes Taken from netty

Figure 8 A change pattern from T5.

Table 5 Statistics about commits and time overhead

	Total commits (#)	Commits after H1 (#)	Commits after (#) H2 and H3	Time overhead (s)			
				Step 1	Step 2	Step 3	Total
Sum	3191057	715620	320222	945.6	257946.2	46521.7	305413.5
Average	2746	616	276	0.8	222.0	40.0	262.8

4.4 Application to bug detection (RQ3)

A direct application of CPAM is to derive bug patterns from the identified change patterns by CPAM. For example, from extracted changes in change pattern ⟨`StringBuffer.append`, `StringBuilder.append`⟩, we can easily derive a bug pattern as "usage of `StringBuffer.append` in single-thread applications is inefficient". Such bug patterns can be useful for enriching bug patterns in bug detectors and detecting new bugs.

4.4.1 *Enriching bug patterns*

To evaluate whether CPAM can help find new bug patterns, we select three state-of-the-art static bug detectors, and search the list of bug patterns that are supported by the detectors to decide whether any detectors have already supported the API misuse patterns reflected in the change patterns in Tables 2–4. In particular, we select SpotBugs (S.B.) (i.e., the successor of FindBugs [54]), Google's Error Prone (E.P.) [55] and PMD [56] because (1) they are actively developed and maintained, (2) they document a list of bug patterns covering a broad range of bugs (e.g., bad practices, multithreading, performance and security), and (3) they are picked as representative detectors in recent studies about static bug detectors [57, 58].

The results are listed in the last three columns in Tables 2–4. Surprisingly, only one bug patterns from SpotBugs, five bug patterns from Error Prone, and zero bug pattern from PMD support the API misuse patterns reflected in our identified top ten change patterns. This finding is similarly observed in a recent study [58] where only 27 of 594 real-world Java bugs can be detected by static bug detectors. The main reason for such a low overlapping is that our identified change patterns often capture logic bugs, which are difficult to detect by static analysis (e.g., AST-based matching or data-flow analysis) employed in the detectors (see more discussion in Subsection 4.5). This indicates that there is a large gap between the bug patterns in existing bug detectors and in real-world bugs, and there is a significant potential to improve bug detection by deriving and supporting new bug patterns from change patterns of API misuses.

4.4.2 *Detecting new bugs*

To evaluate the feasibility of applying CPAM to helping detect new bugs, we select four change patterns (as shown in Table 6). Three are from the top change patterns in Tables 3 and 4, and one is the interesting change pattern of T5 in Figure 8. The first pattern is realized by AST-matching to check whether the

Table 6 Statistics about detected bugs

Change pattern	Bugs	Fixed
⟨Integer.valueOf(String), Integer.parseInt(String)⟩	15	4
⟨String.equals(Object), Objects.equals(Object, Object)⟩	7	4
⟨Thread.sleep(long), CountDownLatch.await(long, TimeUnit)⟩	3	1
StringBuilder.append(String) in T5	19	9
Sum	44	18

return value of `Integer.valueOf(String)` is assigned to an `int` but not `Integer` variable. The second is realized by soot-based data-flow analysis to determine whether the receiver of `String.equals(Object)` can be null. The third pattern is realized through AST-matching to check whether `Thread.sleep(long)` is in both a loop and synchronization structure. The last is realized via AST-matching to decide whether `StringBuilder.append(String)` is consecutively called for more than 5 times. Finally, we discover 44 new bugs with no false positive, and submit bug reports. At the time of writing, 18 have been confirmed and fixed, while the others are still waiting for confirmation.

Specifically, we found 7 null pointer dereferencing bugs, and 37 performance bugs. They are find across 19 open-source projects on GitHub, whose stars rang from 82 to 1093 with an average of 746, and some of which are from companies like Google Cloud, Apache, LinkedIn and Teradata. More details are available at our website[2)]. Interestingly, the bug fixers do not know the difference between `Integer.valueOf` and `Integer.parseInt`; and after they learn the difference from our bug report, they also fix it in their other projects. Similarly, the bug fixers do not believe that chained calls to `append` would have different performance from consecutive calls to `append`; and after we provid benchmarking data about the performance difference, they quickly fix it. A bug fixer from Google Cloud appraise that our change pattern suggest a good practice to avoid null pointer dereferencing bugs in string comparisons. These bug results and discussion from bug reports demonstrate that Cpam is helpful to detect new API misuses and provide fixing suggestions.

Cpam can facilitate bug detection (e.g., enriching bug patterns and detecting new bugs). We have discovered 44 new bugs, 18 of which have been confirmed and fixed.

4.5 Discussion

Threat. The main threats to the validity of our evaluation are twofold. First, we only analyze part of the identified change patterns of API misuses to evaluate Cpam's effectiveness. However, it will be too costly to analyze all change patterns which involve a large number (i.e., 12508) of detected changes to fix API misuses. Instead, we investigate some change patterns for each change type, covering 1262 extracted changes to API misuses. We are continuing to analyze other change patterns to have a more comprehensive understanding of API misuses. Second, we only evaluate Cpam against Java SE APIs. Further studies are required to apply Cpam to more framework or third-party library APIs (e.g., Android or Spring framework APIs).

Limitation. Our heuristic-based selection of commits might miss bug-fix commits or falsely include non-bug-fix commit. As a result, for the missed bug-fix commits, they will contain some API misuses that are missed by Cpam. Using a large corpus of projects, as we do in our evaluation, may partially mitigate this problem and make Cpam still detect common patterns. We also plan to extend recent advances on identifying security commits (e.g., [44, 59, 60]) to accurately identify bug-fix commits.

For the falsely labeled bug-fix commits, they will contain API usage changes that are not related to API misuses. Besides, even for bug-fix commits, they could also contain API usage changes that are not related to API misuses. Our mitigation is to choose bug-fix commits that have small code changes, and the resulting falsely-detected changes of API misuses are around 11.4% for our analyzed change patterns. Another possible mitigation is to use fault localization technique [61] to locate faulty API usages. However, it often needs to run failing and/or passing test cases, which seems not practical for every selected commit. However, it is still worthwhile to apply fault localization to the bug-fix commits that have test cases to investigate its cost and benefits.

False positives are caused by incorrect mapping, refactoring, and tangled code changes, as indicated by our evaluation. Because GumTree has been the state-of-the-art to produce the mapping, we plan to develop several heuristics to improve GumTree by investigating the data of incorrect mappings.

2) https://cpam2019.wixsite.com/mysite.

Moreover, we are integrating the technique in [62] to identify refactoring and the techniques in [47,63,64] to untangle irrelevant code changes.

The derived change types are not meant to be exhaustive, but to have a reasonable coverage of API usage changes to identify representative API misuses, which has been demonstrated in our evaluation. Still, 2119 API usage changes are not captured by CPAM in our evaluation. Our preliminary analysis indicates that incorrect mapping in GUMTREE is one major reason of these missed API usage changes; i.e., an update action to a statement is falsely identified as a delete action and an insert action. Moreover, we also plan to closely investigate the data for potential extensions to our derived change types.

Application. Our evaluation has indicated the feasibility of deriving bug patterns from change patterns for bug detection. At current stage, bug patterns are manually derived, as is similarly done by Paletov et al. [65] to manually derive crypto API rules from code changes. Hence, a next step of CPAM is to automatically extract bug patterns from change patterns. However, it is challenging because of the various root causes of API misuses, especially for logic-related API misuses.

Our identified change patterns often capture logic bugs that are subtle and very difficult to detect because of the various root causes related to the program semantics. However, some logic bug patterns do exist with respect to the semantics of code elements. For example, the semantic similarity between argument names and parameter names of a method is used to detect bugs caused by passing a wrong argument to a method call [49] or passing the arguments in the wrong order when calling a method with multiple parameters of the same type [50–52]. Similarly, we also find such logic bug patterns in our identified change patterns (as illustrated in Figures 1(a), 6(d) and 7(a)). In that sense, CPAM can provide hints on summarizing logic bug patterns, and thus facilitate the detection of logic bugs, which is not supported by existing static bug detectors. One potential approach is to leverage deep learning and knowledge graph to analyze the semantic gap between API calls and their surrounding program elements (e.g., variable or method name). If the semantic gap is large (e.g., Figures 1(a), 6(d) and 7(a)), there is a high potential that APIs are misused.

Last but not the least, we have released the detected changes to fix API misuses from 1162 projects at our website[3)] to the research community to foster applications of this dataset. This dataset is a complement to the API misuse dataset in [66] that focus on the context (e.g., order and pre-condition) of API calls but not the composing elements of a single API call.

5 Related work

Given the large body of work on mining software repositories, we discuss the most relevant work in six aspects, i.e., snapshot-based and history-based API usage mining, API usage applications, change pattern extraction, API documentation directives, and API evolution.

Snapshot-based API usage mining. Project snapshots have been widely used to identify API usage patterns [5]. Since the seminal work of Engler et al. [67], a number of advances have been made. Specifically, Li and Zhou [22] used frequent itemset mining to detect sets of API calls. Thummalapenta and Xie [23] detected both frequent and infrequent alternative patterns of an API call to reduce false positives in their violations. Salman [68] used formal concept analysis to distinguish the set of APIs that are always or frequently used together. Monperrus et al. [24] mined type-usages, i.e., a set of API calls on a variable of a type. Liang et al. [33] reduced noise in the code for reducing false negatives and false positives of the violations of mined patterns.

To consider the sequential order among the APIs in the mined patterns, Xie and Pei [69] used frequent subsequence mining to identify sequential API usage patterns, followed by a number of advances. Kagdi et al. [70] further detected control structures surrounding each API call. Acharya et al. [71] detected API patterns as partial order graphs to provide general and concise API ordering information. Wasylkowski et al. [25] mined sequential type-usages. Zhong et al. [6] integrated clustering to mine API usage patterns sensitive to programming contexts. Gruska et al. [72] mined temporal properties that capture the order, in which values flow through API calls. Thummalapenta and Xie [73] detected API usage patterns in exception paths. Wang et al. [74] mined succinct and high-coverage API usage patterns. Moritz et al. [26] mined usage patterns that occur across several functions. Fowkes and Sutton [27] applied a probabilistic model to mine the most interesting API call patterns. Gu et al. [75] used deep learning to generate API usage sequences for a natural language query. Zhang et al. [28] mined order and guard

3) https://cpam2019.wixsite.com/mysite.

condition of API calls. Wen et al. [76] mined API misuse patterns via mutation analysis. Moreover, several approaches [38, 39, 77–80] modeled API usages as graphs using graph-based algorithms. n-gram model [81] and statistical model [82] were used to identify low-probability API sequences as potential bugs. Several studies [34–37] mined guard conditions of API calls.

These approaches detected sets or sequences of API calls or guard conditions of API calls from code snapshots, which can be considered as the mining of the context of API calls. Instead, we detect the change patterns of the single API call itself from revision history, which is complementary to them.

History-based API usage mining. History-based techniques use source code changes between historical revisions. Williams and Hollingsworth [53] mined functions whose return value is checked in added code, and designed a return value checker to find potential bugs (i.e., the return value of mined functions is used before being checked). They focus on the change type T4, while we target the change types T1–T3. The most closely related studies are [65, 83]. Murphy-Hill et al. [83] analyzed patches to identify API usability issues at Google. However, they cannot detect the change type T1, and no technical detail is presented in their four-page paper. Paletov et al. [65] mined crypto API usage changes from code changes, and manually inferred 13 rules. While they focus on crypto APIs, we try to solve the general problem of detecting API misuse patterns.

Williams and Hollingsworth [29] also mined function pairs that were frequently added together in revision history. Livshits and Zimmermann [30] improved [29] by mining function sequences on the same object as well as nested patterns of function pairs and function sequences. Uddin et al. [31, 84] mined temporal API usage patterns, where a pattern contained a set of concepts (i.e., distinct groups of APIs that were added together in the revision history) added in a temporal order. Azad et al. [32] mined co-occurring API calls from revision history of apps and from Stack Overflow posts. These approaches, except for [32], focus on system-specific patterns within a single project, while we aim at common patterns across projects. Further, we focus on change patterns to the composing elements of an API call but not the patterns of its context. To the best of our knowledge, this is the first study to systematically detect such patterns.

API usage applications. API usage patterns could be used in various applications. A few examples include detecting pattern violations to detect bugs (e.g., [23,30,53,81,82]), suggesting code completion and recommendation (e.g., [6, 85]), enhancing documentation (e.g., [31, 84]), generating API usage examples (e.g., [27, 86]) and finding API misuses in Stack Overflow posts [28]. Our change patterns are mainly for facilitating bug detection or providing programming practice advice for developers.

Change pattern extraction. Several studies extract change patterns as syntactic program transformations from code changes to support various tasks, e.g., programming activities [87], repetitive code edits [88–90], program repairing [90–94] and mutation operators [95]. They need more efforts to ensure syntactic correctness after applying transformations. Differently, Cpam is to detect change patterns of API misuses, and need more efforts to identify which part of an API call is misused.

API documentation directives. API documentation directives are natural language statements in API documentation that make developers aware of the constraints and guidelines about API usages [96]. Dekel and Herbsleb [97] implemented an Eclipse plug-in to decorate method calls whose targets have associated directives. Monperrus et al. [96] identified the significance of API documentation directives, and provided a taxonomy of 23 kinds of directives. Then, Saied et al. [98] and Zhou et al. [99] targeted four of the method call directives, and proposed techniques to detect directive defects in documentation, assuming that the API usage code is correct. Differently, we can detect the patterns of violating method call directives from code changes. We can also detect other patterns that are not related to method call directives, e.g., API replacements.

API evolution. We study API usage changes, whereas many researchers study API evolution. For example, Dig and Johnson [10] provided a list of breaking API changes; Jezek et al. [12], Raemaekers et al. [13] and Xavier et al. [11] studied the impact of API evolution on client programs; and Wu et al. [100], Dagenais and Robillard [101] detected the change rules of APIs to recommend adaptive changes to client programs.

6 Conclusion

In this paper, we have proposed Cpam to detect change patterns of API misuses from the code changes of a corpus of open-source projects. We have implemented Cpam for Java, and conducted large-scale

experiments using 1162 Java projects. Our experimental results demonstrate the effectiveness and efficiency of Cpam in detecting change patterns of API misuses. We have discussed some interesting change patterns and applied some of them for bug detection, discovering 44 new bugs while 18 of them have been confirmed and fixed. In the future, we plan to investigate logic bug detection.

Acknowledgements This work was supported by National Natural Science Foundation of China (Grant No. 61802067).

References

1 Robillard M P, DeLine R. A field study of API learning obstacles. Empir Softw Eng, 2011, 16: 703–732
2 Hou D, Li L. Obstacles in using frameworks and APIs: an exploratory study of programmers' newsgroup discussions. In: Proceedings of the 2011 IEEE 19th International Conference on Program Comprehension, 2011. 91–100
3 Nadi S, Krüger S, Mezini M, et al. Jumping through hoops: why do Java developers struggle with cryptography apis? In: Proceedings of the 38th International Conference on Software Engineering, 2016. 935–946
4 Zibran M F, Eishita F Z, Roy C K. Useful, but usable? factors affecting the usability of APIs. In: Proceedings of the 2011 18th Working Conference on Reverse Engineering, 2011. 151–155
5 Robillard M P, Bodden E, Kawrykow D, et al. Automated API property inference techniques. IEEE Trans Softw Eng, 2013, 39: 613–637
6 Zhong H, Xie T, Zhang L, et al. MAPO: mining and recommending API usage patterns. In: Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming, 2009. 318–343
7 Uddin G, Robillard M P. How API documentation fails. IEEE Softw, 2015, 32: 68–75
8 Linares-Vásquez M, Bavota G, Bernal-Cárdenas C, et al. API change and fault proneness: a threat to the success of android apps. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, 2013. 477–487
9 McDonnell T, Ray B, Kim M. An empirical study of API stability and adoption in the android ecosystem. In: Proceedings of the 2013 IEEE International Conference on Software Maintenance, 2013. 70–79
10 Dig D, Johnson R. How do APIs evolve? A story of refactoring. J Softw Maint Evol-Res Pract, 2006, 18: 83–107
11 Xavier L, Brito A, Hora A, et al. Historical and impact analysis of API breaking changes: a large-scale study. In: Proceedings of 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), 2017. 138–147
12 Jezek K, Dietrich J, Brada P. How Java APIs break — an empirical study. Inf Softw Tech, 2015, 65: 129–146
13 Raemaekers S, van Deursen A, Visser J. Semantic versioning and impact of breaking changes in the Maven repository. J Syst Softw, 2017, 129: 140–158
14 Jung C, Rus S, Railing B P, et al. Brainy: effective selection of data structures. SIGPLAN Not, 2011, 46: 86–97
15 Xu G. CoCo: sound and adaptive replacement of Java collections. In: Proceedings of the 27th European conference on Object-Oriented Programming, 2013. 1–26
16 Chen B, Liu Y, Le W. Generating performance distributions via probabilistic symbolic execution. In: Proceedings of the 38th International Conference on Software Engineering, 2016. 49–60
17 Zhao Y, Xiao L, Wang X, et al. Localized or architectural: an empirical study of performance issues dichotomy. In: Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), 2019. 316–317
18 Georgiev M, Iyengar S, Jana S, et al. The most dangerous code in the world: validating SSL certificates in non-browser software. In: Proceedings of the 2012 ACM Conference on Computer and Communications Security, 2012. 38–49
19 Fahl S, Harbach M, Perl H, et al. Rethinking SSL development in an appified world. In: Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security, 2013. 49–60
20 Egele M, Brumley D, Fratantonio Y, et al. An empirical study of cryptographic misuse in android applications. In: Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security, 2013. 73–84
21 Li L, Bissyandé T F, Traon Y L, et al. Accessing inaccessible android APIs: an empirical study. In: Proceedings of the 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2016. 411–422
22 Li Z, Zhou Y. PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. SIGSOFT Softw Eng Notes, 2005, 30: 306–315
23 Thummalapenta S, Xie T. Alattin: mining alternative patterns for detecting neglected conditions. In: Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering, 2009. 283–294
24 Monperrus M, Bruch M, Mezini M. Detecting missing method calls in object-oriented software. In: Proceedings of the 24th European Conference on Object-Oriented Programming, 2010. 2–25
25 Wasylkowski A, Zeller A, Lindig C. 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, 2007. 35–44
26 Moritz E, Linares-Vásquez M, Poshyvanyk D, et al. ExPort: detecting and visualizing API usages in large source code repositories. In: Proceedings of the 2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2013. 646–651
27 Fowkes J, Sutton C. Parameter-free probabilistic API mining across github. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2016. 254–265
28 Zhang T, Upadhyaya G, Reinhardt A, et al. Are code examples on an online Q&A forum reliable? a study of API misuse on stack overflow. In: Proceedings of 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE), 2018. 886–896
29 Williams C C, Hollingsworth J K. Recovering system specific rules from software repositories. SIGSOFT Softw Eng Notes, 2005, 30: 1
30 Livshits B, Zimmermann T. DynaMine: finding common error patterns by mining software revision histories. SIGSOFT Softw Eng Notes, 2005, 30: 296–305
31 Uddin G, Dagenais B, Robillard M P. Temporal analysis of API usage concepts. In: Proceedings of the 2012 34th International Conference on Software Engineering (ICSE), 2012. 804–814
32 Azad S, Rigby P C, Guerrouj L. Generating API call rules from version history and stack overflow posts. ACM Trans Softw Eng Methodol, 2017, 25: 1–22
33 Liang B, Bian P, Zhang Y, et al. Antminer: mining more bugs by reducing noise interference. In: Proceedings of the 38th International Conference on Software Engineering, 2016. 333–344

34 Ramanathan M K, Grama A, Jagannathan S. Path-sensitive inference of function precedence protocols. In: Proceedings of the 29th International Conference on Software Engineering (ICSE'07), 2007. 240–250
35 Nguyen H A, Dyer R, Nguyen T N, et al. Mining preconditions of APIs in large-scale code corpus. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2014. 166–177
36 Ramanathan M K, Grama A, Jagannathan S. Static specification inference using predicate mining. SIGPLAN Not, 2007, 42: 123–134
37 Wasylkowski A, Zeller A. Mining temporal specifications from object usage. Autom Softw Eng, 2011, 18: 263–292
38 Chang R Y, Podgurski A, Yang J. Finding what's not there: a new approach to revealing neglected conditions in software. In: Proceedings of the 2007 International Symposium on Software Testing and Analysis, 2007. 163–173
39 Nguyen T T, Nguyen H A, Pham N H, et al. Graph-based mining of multiple object usage patterns. In: Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, 2009. 383–392
40 Falleri J R, Morandat F, Blanc X, et al. Fine-grained and accurate source code differencing. In: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering, 2014. 313–324
41 Kim S, Whitehead E J, Zhang Y. Classifying software changes: clean or buggy? IEEE Trans Softw Eng, 2008, 34: 181–196
42 Jin G, Song L, Shi X, et al. Understanding and detecting real-world performance bugs. SIGPLAN Not, 2012, 47: 77–88
43 Chen Z, Chen B, Xiao L, et al. Speedoo: prioritizing performance optimization opportunities. In: Proceedings of the 40th International Conference on Software Engineering, 2018. 811–821
44 Zhou Y, Sharma A. Automated identification of security issues from commit messages and bug reports. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, 2017. 914–919
45 Wei L, Liu Y, Cheung S C. Taming android fragmentation: characterizing and detecting compatibility issues for Android apps. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, 2016. 226–237
46 Herzig K, Zeller A. The impact of tangled code changes. In: Proceedings of the 2013 10th Working Conference on Mining Software Repositories (MSR), 2013. 121–130
47 Dias M, Bacchelli A, Gousios G, et al . Untangling fine-grained code changes. In: Proceedings of the 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER), 2015. 341–350
48 Hattori L P, Lanza M. On the nature of commits. In: Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering-Workshops, 2008. 63–71
49 Liu H, Liu Q, Staicu C A, et al. Nomen est omen: exploring and exploiting similarities between argument and parameter names. In: Proceedings of the 38th International Conference on Software Engineering, 2016. 1063–1073
50 Pradel M, Gross T R. Detecting anomalies in the order of equally-typed method arguments. In: Proceedings of the 2011 International Symposium on Software Testing and Analysis, 2011. 232–242
51 Pradel M, Gross T R. Name-based analysis of equally typed method arguments. IEEE Trans Softw Eng, 2013, 39: 1127–1143
52 Rice A, Aftandilian E, Jaspan C, et al. Detecting argument selection defects. In: Proceedings of the ACM on Programming Languages, 2017. 1–22
53 Williams C C, Hollingsworth J K. Automatic mining of source code repositories to improve bug finding techniques. IIEEE Trans Softw Eng, 2005, 31: 466–480
54 Hovemeyer D, Pugh W. Finding bugs is easy. SIGPLAN Not, 2004, 39: 92–106
55 Aftandilian E, Sauciuc R, Priya S, et al. Building useful program analysis tools using an extensible Java compiler. In: Proceedings of the 2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation, 2012. 14–23
56 Copeland T. PMD Applied. Arexandria: Centennial Books, 2005
57 Thung F, Lucia F, Lo D, et al. To what extent could we detect field defects? An extended empirical study of false negatives in static bug-finding tools. Autom Softw Eng, 2015, 22: 561–602
58 Habib A, Pradel M. How many of all bugs do we find? a study of static bug detectors. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, 2018. 317–328
59 Sabetta A, Bezzi M. A practical approach to the automatic classification of security-relevant commits. In: Proceedings of 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2018. 579–582
60 Xu Z, Chen B, Chandramohan M, et al. SPAIN: security patch analysis for binaries towards understanding the pain and pills. In: Proceedings of the 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), 2017. 462–472
61 Pearson S, Campos J, Just R, et al. Evaluating and improving fault localization. In: Proceedings of the 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), 2017. 609–620
62 Kawrykow D, Robillard M P. Non-essential changes in version histories. In: Proceedings of the 2011 33rd International Conference on Software Engineering (ICSE), 2011. 351–360
63 Barnett M, Bird C, Brunet J A, et al. Helping developers help themselves: automatic decomposition of code review changesets. In: Proceedings of the 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, Florence, 2015. 134–144
64 Tao Y, Kim S, Partitioning composite code changes to facilitate code review. In: Proceedings of the 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories, 2015. 180–190
65 Paletov R, Tsankov P, Raychev V, et al. Inferring crypto API rules from code changes. SIGPLAN Not, 2018, 53: 450–464
66 Amann S, Nguyen H A, Nadi S, et al. A systematic evaluation of static API-misuse detectors. IEEE Trans Softw Eng, 2019, 45: 1170–1188
67 Engler D, Chen D Y, Hallem S, et al. Bugs as deviant behavior: a general approach to inferring errors in systems code. SIGOPS Oper Syst Rev, 2001, 35: 57–72
68 Salman H E. Identification multi-level frequent usage patterns from APIs. J Syst Softw, 2017, 130: 42–56
69 Xie T, Pei J. MAPO: mining API usages from open source repositories. In: Proceedings of the 2006 International Workshop on Mining Software Repositories, 2006. 54–57
70 Kagdi H, Collard M L, Maletic J I. An approach to mining call-usage patternswith syntactic context. In: Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering, 2007. 457–460
71 Acharya M, Xie T, Pei J, et al. Mining API patterns as partial orders from source code: from usage scenarios to specifications. In: Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, 2007. 25–34
72 Gruska N, Wasylkowski A, Zeller A. Learning from 6000 projects: lightweight cross-project anomaly detection. In: Proceedings of the 19th International Symposium on Software Testing and Analysis, 2010. 119–130

73 Thummalapenta S, Xie T. Mining exception-handling rules as sequence association rules. In: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering, 2009. 496–506
74 Wang J, Dang Y, Zhang H, et al. Mining succinct and high-coverage API usage patterns from source code. In: Proceedings of the 2013 10th Working Conference on Mining Software Repositories (MSR), 2013. 319–328
75 Gu X, Zhang H, Zhang D, et al. Deep API learning. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2016. 631–642
76 Wen M, Liu Y, Wu R, et al. Exposing library API misuses via mutation analysis. In: Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), 2019
77 Mandelin D, Xu L, Bodík R, et al. Jungloid mining: Helping to navigate the API jungle. SIGPLAN Not, 2005, 40: 48–61
78 Zhong H, Zhang H L, Mei H. Inferring specifications of object oriented APIs from API source code. In: Proceedings of the 2008 15th Asia-Pacific Software Engineering Conference, 2008. 221–228
79 Buse R P, Weimer W. Synthesizing API usage examples. In: Proceedings of the 2012 34th International Conference on Software Engineering (ICSE), 2012. 782–792
80 Niu H, Keivanloo I, Zou Y. API usage pattern recommendation for software development. J Syst Softw, 2017, 129: 127–139
81 Wang S, Chollak D, Movshovitz-Attias D, et al. Bugram: bug detection with n-gram language models. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, 2016. 708–719
82 Murali V, Chaudhuri S, Jermaine C. Bayesian specification learning for finding API usage errors. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, 2017. 151–162
83 Murphy-Hill E, Sadowski C, Head A, et al. Discovering API usability problems at scale. In: Proceedings of the 2nd International Workshop on API Usage and Evolution, 2018. 14–17
84 Uddin G, Dagenais B, Robillard M P. Analyzing temporal API usage patterns. In: Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), 2011. 456–459
85 Bruch M, Monperrus M, Mezini M. Learning from examples to improve code completion systems. In: Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, 2009. 213–222
86 Wang L, Fang L, Wang L, et al. APIExample: an effective web search based usage example recommendation system for Java APIs. In: Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), 2011. 592–595
87 Negara S, Codoban M, Dig D, et al. Mining fine-grained code changes to detect unknown change patterns. In: Proceedings of the 36th International Conference on Software Engineering, 2014. 803–813
88 Meng N, Kim M, McKinley K S. Systematic editing: generating program transformations from an example. SIGPLAN Not, 2011, 46: 329–342
89 Meng N, Kim M, McKinley K S. LASE: locating and applying systematic edits by learning from examples. In: Proceedings of the 2013 35th International Conference on Software Engineering (ICSE), 2013. 502–511
90 Rolim R, Soares G, D'Antoni L, et al. Learning syntactic program transformations from examples. In: Proceedings of the 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), 2017. 404–415
91 Kim D, Nam J, Song J, et al. Automatic patch generation learned from human-written patches. In: Proceedings of the 2013 35th International Conference on Software Engineering (ICSE), 2013. 802–811
92 Long F, Amidon P, Rinard M. Automatic inference of code transforms for patch generation. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, 2017. 727–739
93 Liu X, Zhong H. Mining stackoverflow for program repair. In: Proceedings of the 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), 2018. 118–129
94 Roychoudhury A, Xiong Y. Automated program repair: a step towards software automation. Sci China Inf Sci, 2019, 62: 200103
95 Brown D B, Vaughn M, Liblit B, et al. The care and feeding of wild-caught mutants. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, 2017. 511–522
96 Monperrus M, Eichberg M, Tekes E, et al. What should developers be aware of? An empirical study on the directives of API documentation. Empir Software Eng, 2012, 17: 703–737
97 Dekel U, Herbsleb J D. Improving API documentation usability with knowledge pushing. In: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering, 2009. 320–330
98 Saied M A, Sahraoui H, Dufour B. An observational study on API usage constraints and their documentation. In: Proceedings of 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER), 2015. 33–42
99 Zhou Y, Gu R, Chen T, et al. Analyzing APIs documentation and code to detect directive defects. In: Proceedings of 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), 2017. 27–37
100 Wu W, Guéhéneuc Y G, Antoniol G, et al. AURA: a hybrid approach to identify framework evolution. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, 2010. 325–334
101 Dagenais B, Robillard M P. Recommending adaptive changes for framework evolution. ACM Trans Softw Eng Methodol, 2011, 20: 1–35