Are Online Code Examples Reliable? An Empirical Study of API Misuse on Stack Overflow

***Abstract*—Programmers often search for code examples to learn new APIs. However, there has been little research on the reliability of online code examples—whether a code example may lead to unexpected behavior when it is reused verbatim. This paper presents an empirical study that assesses the reliability of code snippets appearing in an online Q&A forum, Stack Overflow via API misuse detection. To reduce manual assessment effort, we design and implement MAPLE, an API usage mining approach that extracts patterns from over 7 million GitHub projects and subsequently reports potential API usage violations in Stack Overflow posts. MAPLE infers API usage patterns accurately with 80% precision and 91% recall in top 5 patterns.**

We analyze 31,801 Stack Overflow posts using MAPLE and find that over 50% of them have potential API usage violations that may produce the symptoms of program crashes and resource leaks. Such API misuse is caused by three main reasons—*missing control constructs*, *missing or incorrect order of API calls*, and *incorrect guard conditions*. We observe that recognized posts, even those accepted as correct answers or upvoted by other programmers, are not necessarily more reliable than unrecognized ones in terms of API misuse. This study demonstrates the prevalence and severity of API misuse on Stack Overflow and calls for a new human-in-the-loop approach to enrich and enhance online code examples in a systematic and structured manner using mined API usage patterns.

I. INTRODUCTION

Programmers often resort to online code examples during software development [1, 2]. A case study at Google shows that developers issue an average of 12 code search queries per weekday [2]. Online code examples have also been utilized to facilitate other development activities, e.g, bug fixing [3], debugging [4], and API documentation [5]–[7].

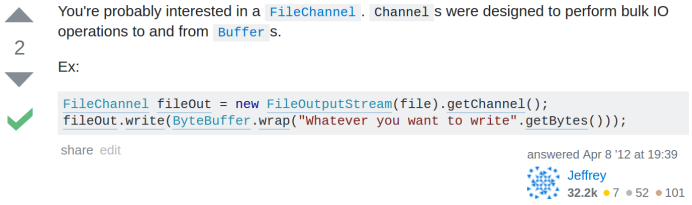
Previous studies have analyzed the quality of code snippets appearing in Q&A forums from different perspectives. Both Subramanian et al. [8] and Yang et al. [9] show that the majority of code snippets on Stack Overflow are free-standing program statements that cannot be accepted by compilers. Dagenais and Robillard [10] find that 89% of APIs mentioned in online forums cannot be easily resolved due to the ambiguity of API names. Zhou et al. [11] find that 86 of 200 posts that are accepted as correct answers still use deprecated APIs but only 3 of them are acknowledged on Stack Overflow. None of these studies have investigated the reliability of online code examples in the sense that following a code example verbatim may lead to API usage violations in a target application. Currently, there is no tool support to help developers easily recognize and repair unreliable code snippets in online Q&A forums by leveraging API usage patterns mined from massive code corpora.

This paper aims to assess the reliability of Stack Overflow posts by contrasting their code snippets against desirable API usage patterns mined from over 7 million GitHub projects. Our hypothesis is that commonly recurring API usage from massive corpora may represent a desirable pattern that a programmer can use to trust or enhance code snippets on Stack Overflow.

Existing API usage mining techniques learn from either a single project [12] or a small code corpus [13]–[20] and do not scale to the size of millions of software projects. Compared with the use of such in-house small corpus, mining massive corpora can provide more confidence to a programmer about whether to trust an online code example as is. Our goal is to scale our API usage mining technology to massive corpora without sacrificing the *fidelity* and *expressiveness* of the API usage representation. To achieve this goal, we first implement control and data flow analysis on top of Boa's ultra-large-scale software mining infrastructure [21], so that we can use program slicing to ignore irrelevant API calls. Second, we combine frequent subsequence mining and SMT-based guard condition mining so that we can retain both the temporal ordering of related API calls and appropriate guard conditions for each API call.

MAPLE efficiently searches over 7 million GitHub projects and retrieves an average of 32,678 code snippets for a given API within 10 min. Using confirmed patterns from an existing benchmark [22] and hand-coded patterns based on JavaDoc documentation as the ground truth, we find that MAPLE infers API usage patterns from the massive corpora with 80% precision and 91% recall, when considering top 5 patterns. In contrast, MAPLE infers patterns with 49% precision and 46% recall from a small code corpus with 7,988 GitHub projects. Our results indicate that learning from massive code corpora can effectively filter API usage patterns pertinent to a few outlier code snippets and therefore significantly improve the precision and recall of API usage mining.

Using this data set of API usage patterns found by MAPLE and further validated by the first author, we study the extent of API misuse in Stack Overflow posts. More than half of the SO posts in our study contain API misuse that could produce the symptoms of program crashes and resource leaks. Such API misuse is caused by three main reasons—*missing control constructs*, *missing or incorrect order of API calls*, and *incorrect guard conditions*. API misuse appears in 51% of recognized posts vs. 50% unrecognized posts. Within recognized posts, we do not observe a strong positive or negative correlation between the vote scores (i.e., upvotes minus downvotes) and



(a) An example that does not close `FileChannel` properly

(b) An example that misses exception handling

Fig. 1: Two code examples about how to write data to a file using `FileChannel` on Stack Overflow

the number of posts with API usage violations.

This observation indicates that code examples accepted as correct answers or endorsed by other programmers are not necessarily more reliable than unrecognized ones. Our study demonstrates the prevalence and severity of API misuse in online Q&A posts and calls for a new human-in-the-loop approach to check and augment online code examples by leveraging API usage mined from massive corpora.

II. Motivating Examples

This section motivates our study with a code reuse scenario using Stack Overflow. Suppose Alice wants to write data to a file using `FileChannel`. Alice searches on Stack Overflow and finds two code examples, as shown in Figure 1.[1][2] Both examples are accepted as correct answers and also upvoted by other programmers. However, both of them have potential API usage violations that may induce unexpected behavior.

The first post in Figure 1a does not call `FileChannel.close` to close the channel. If Alice copies this example to a program that does not heavily access new file resources, this example may behave properly because OS will clean up unmanaged file resources eventually after the program exits. However, if Alice reuses the example in a long-running program with heavy IO, such lingering file resources may cause file handle leaks. Since most operating systems limit the number of opened files, unclosed file streams can eventually run out of file handle resources. If Alice uses `FileChannel` to write a big volume of data, she may also lose cached data in the underlying file output stream, if she forgets to flush or close the channel.

Even though the second example in Figure 1b calls `FileChannel.close`, it does not handle the potential exceptions thrown by `FileChannel.write`. For example, calling this method would throw `ClosedChannelException`, if the channel is already closed. If Alice uses `FileChannel` in a concurrent program where multiple threads attempt to access the same channel, `AsynchronousCloseException` will occur if one thread closes the channel, while another thread is still writing data to the channel.

As a novice programmer, Alice cannot easily recognize such API misuse in these posts. In this case, our approach MAPLE infers desirable API usage by scanning over 7 million GitHub projects and by finding 3,277 Java methods that also call `FileChannel.write`. This API usage mining process finds two commonly used patterns. The mostly frequent usage supported by 1425 code snippets on GitHub indicates that a method call to `FileChannel.write()` must be contained inside a try and catch block. The second most frequent usage supported by 1245 code snippets on GitHub indicates that `FileChannel.write` must be followed by `FileChannel.close`. By comparing code snippets in Figures 1a and 1b against these two API usage patterns, MAPLE could warn the user the potential risk of using the code snippets verbatim, and suggest her to consider adding a missing call to `close` and an exception handling block.

$$
\begin{aligned}
\mathit{sequence} &:= \epsilon \mid \mathit{call}\ ;\ \mathit{sequence} \\
&\quad \mid \mathit{construct}\ \{\ ;\ \mathit{sequence}\ ;\ \}\ ;\ \mathit{sequence} \\
\mathit{call} &:= \mathit{name}(n)@\mathit{condition} \\
\mathit{construct} &:= \text{if} \mid \text{else} \mid \text{loop} \mid \text{try} \mid \text{catch} \mid \text{finally} \\
\mathit{condition} &:= \text{boolean expression} \\
\mathit{name} &:= \text{characters} \\
n &:= \text{integer constant}
\end{aligned}
$$

Fig. 2: Grammar of Structured API Call Sequence

III. Pattern Mining and API Misuse Detection

Given an API method of interest, MAPLE takes four phases to infer desirable API usage from massive corpora and to flag potential API misuse in code snippets appearing on Stack Overflow. The first three phases are automated. The last phase allows a human to inspect the inferred API usage patterns first before using them to flag potential violations in the Stack Overflow posts.

In Phase 1, the goal of MAPLE is to extract a sequence of API calls, including a call to a particular API method of interest and to remove irrelevant statements by leveraging program slicing. In Phase 2, MAPLE finds a common subsequence from individual sequences of API calls. In Phase 3, in order to retain conditions under which each API can be invoked, MAPLE mines guard conditions associated with individual API calls. During this process of mining guard conditions, MAPLE uses a SMT solver, Z3 [23], to check the semantic equivalence of guard conditions, so that we can accurately estimate the frequency of each unique guard correctly. In the last Phase 4, MAPLE allows the user to inspect inferred patterns before constrasting them against code snippets in the Stack Overflow posts to report potential API misuse.

[1] http://stackoverflow.com/questions/10065852
[2] http://stackoverflow.com/questions/10506546

```
1  void initInterfaceProperties(String temp, File dDir) {
2   if(!temp.equals("props.txt")) {
3    log.error("Wrong Template.");
4    return;
5   }
6   // load default properties
7   FileInputStream in = new FileInputStream(temp);
8   Properties prop = new Properties();
9   prop.load(in);
10  // init properties
11  prop.set("interface", PROPERTIES.INTERFACE);
12  prop.set("uri", PROPERTIES.URI);
13  prop.set("version", PROPERTIES.VERSION);
14  // write to the property file
15  String fPath=dDir.getAbosulatePath()+"/interface.prop";
16  File file = new File(fPath);
17  if(!file.exists()) {
18   file.createNewFile();
19  }
20  FileOutputStream out = new FileOutputStream(file);
21  prop.store(out, null);
22  in.close();
23 }
```

Fig. 3: A Java method on GitHub that calls `createNewFile`.

A. Structured Call Sequence Extraction and Slicing on GitHub

Given an API method of interest, MAPLE searches individual code snippets that invoke the same API method in the GitHub corpora. Specifically, MAPLE scans 7,830,023 Java projects, collected on September, 2015. To scale code search to millions of software projects, MAPLE leverages a distributed software mining infrastructure, Boa [21] to traverse the abstract syntax trees (ASTs) of Java files in GitHub projects. MAPLE visits every AST method and looks for a method invocation of the API of interest. Figure 3 shows a code snippet retrieved from GitHub for the `File.createNewFile` API. This example creates a property file if it does not exist by calling `createNewFile` (line 18).

To extract the essence of API usage, MAPLE models each code snippet as a structured call sequence, which abstracts away certain syntactic details such variable names and types, but still retains the temporal ordering and guard conditions of API calls in a compact manner. Figure 2 defines the grammar of our API usage representation. A structured call sequence consists of relevant control constructs and API calls, separated by the delimiter ";". API calls are annotated with the number of arguments to distinguish method overloading. Each API call is associated with a guard condition that protects its usage or `true` if it is not guarded by any condition.

MAPLE builds the control flow graph (CFG) of a GitHub snippet and identifies the enclosing control constructs [24]. The enclosing control construct is related to a given API call of interest, if there exists a path between the two and the API call is not post-dominated by the control construct. For instance, the API call to `createNewFile` (line 18) is control dependent on the `if` statements in lines 2 and 17 in Figure 3. From each control construct, we lift the contained predicate. This process is a pre-cursor for mining a guard condition that protects each API method call. We use the conjunction of the lifted predicates in all relevant control constructs. If an API call is in the false branch of a control construct, we negate the predicate when constructing the guard. In Figure 3, since `createNewFile` is in the false branch of the first `if` statement at line 2 and the true branch of the second `if` statement at line 17, its guard condition is `temp.equals("props.txt") && !file.exists()`. While this approach of lifting control predicates does not consider the effect of program statements before an API call via symbolic execution and thus could produce an imprecise guard condition, MAPLE makes this choice of sacrificing precision for scalability. Project-specific predicates and variable names used in the guard conditions are later generalized in Phase 3 to unify equivalent guard conditions regardless of project-specific details.

Bound	Variables	Structured Call Sequence
k=1	file	new File; if {; createNewFile; }; new FileOutputStream
k=2	file, fPath, out	getAbsolutePath; new File; if {; createNewFile; }; new FileOutputStream; store
k=3	file, fPath, out, prop	new Properties; load; set; set; set; getAbsolutePath; new File; if {; createNewFile; }; new FileOutputStream; store
k=∞	file, fPath, out, prop, in, temp	new FileInputStream; new Properties; load; set; set; set; getAbsolutePath; new File; if {; createNewFile; }; new FileOutputStream; store; close
No Slicing	file, fPath, out, prop, in, temp, log	if {; debug; }; new FileInputStream; getAbsolutePath; load; set; set; set; new File; if {; createNewFile; }; new Properties; new FileOutputStream; store; close

TABLE I: Structured call sequences sliced using k bounds. The guard conditions are omitted for the presentation purpose.

Our insight is that MAPLE should filter any statements that are not related to the API method of interest. For example, API calls related to `Properties` in Figure 3 should be removed, since they are project-specific and irrelevant to the invocation of `createNewFile`. MAPLE performs intra-procedural program slicing to retain only data-dependent statements [25]. During this process, MAPLE uses both backward and forward slicing to identify relevant statements up to k hops. Setting k to 1 retains only immediately dependent API calls in the call sequence, while setting k to ∞ includes all transitively dependent API calls. For instance, the `Properties` APIs such as `load` (line 9) and `set` (lines 11-13) are transitively dependent on `createNewFile` through variables `file`, `out`, and `prop`. Table I shows the call sequences extracted from the code snippet in Figure 3 with different k. Setting k to 1 by default leads to best performance and precision empirically (to be detailed in Section VI).

B. Frequent Subsequence Mining

Given a set of structured call sequences from Phase 1, MAPLE finds common subsequences using an efficient sequence mining algorithm, BIDE [26]. Computing the common subsequence is widely practiced in the literature of API usage mining [18, 27]–[29] and this has the benefit of filtering out API calls pertinent to only a few outlier examples. MAPLE splits each structured call sequence by the delimiter ";" and excludes the guard condition of each API call. In other words, MAPLE considers only the ordering of API calls and inclusion of relevant control constructs. The task of mining a common

API Call	Guard	Generalized	Symbolized
s.substring(start)	start>=0 && start<=s.length()	start>=0 && start<=s.length()	arg0>=0 && arg0<=rcv.length()
log.substring(index)	-1<index && index<log.length()+1	-1<index && index<log.length()+1	-1<arg0 && arg0<rcv.length()+1
f.substring(f.indexOf("/"))	dir != null && f.indexOf("/")>=0 && f.indexOf("/")<=f.length()	true && f.indexOf("/")>=0 && f.indexOf("/")<=f.length()	true && arg0>=0 && arg0<=rcv.length()

TABLE II: Example guard conditions of `String.substring`. API Call shows three example call sites. Guard shows the guard condition associated with each call site. Generalized shows the guard conditions after eliminating project-specific predicates. Symbolized shows the guard conditions after symbolizing variable names.

guard condition is done in Phase 3 instead. BIDE mines *frequent closed sequences* above a given minimum support threshold σ. A sequence is a frequent closed sequence, if it occurs frequently above the given threshold and there is no super-sequence with the same support. This algorithm is a good fit to our problem for two reasons. First, it does not need to keep track of already mined subsequence candidates, which significantly reduces the space complexity. Second, it does not mine all frequent patterns but only *closed* ones, leading to a more compact set. MAPLE ranks a list of sequence patterns based on support. MAPLE also filters invalid sequence patterns that do not follow the grammar in Figure 2, as frequent subsequence mining can find invalid patterns with unbalanced brackets such as "`foo@true; }; }`". The default threshold σ is set to 0.5, which means frequent subsequences are reported, only if more than half of relevant code snippets on GitHub include the subsequence.

C. Guard Condition Mining

Given a common subsequence from Phase 2, MAPLE mines the common guard condition of each API call in the sequence. The rationale is that each method call in the common subsequence may have a guard to ensure that the constituent API call does not lead to a failure. Therefore, MAPLE collects all guard conditions from each structured call sequence from Phase 1 and clusters them based on their semantic equivalence. The guard conditions extracted from GitHub code snippets often contain project-specific predicates and variable names. In Figure 3, the identified guard condition of `createNewFile` (line 18) is `temp.equals("props.txt") && !file.exists()`. Its first predicate `temp.equals("props.txt")` checks whether a string variable `temp` contains a specific content. Neither the variable `temp` nor the predicate are related to the usage of `createNewFile`. Therefore, MAPLE first abstracts away such syntactic details in individual guard conditions before clustering them. For each guard condition from Phase 1, MAPLE substitutes predicates that do not mention the receiver object or input arguments of the given API call with `true`. This ensures that the generalized guard condition is still implied by the original guard after removing project-specific predicates. In addition, since each code snippet may use different object and variable names, we normalize these names in the guard conditions. MAPLE uses `rcv` and `argi` as the symbolic names of the receiver and the i-th input argument.

Table II illustrates how we derive the guard condition for `String.substring`. `String.substring` takes an integer index as input and returns a substring that begins from the given index. The third guard condition in Column Guard contains a project-specific predicate, `dir != null`. Since such predicate is not related to `String.substring`, MAPLE substitutes `dir != null` with `true`, as shown in Column Generalized. All three examples name the receiver object differently, i.e., `s`, `log`, and `f`. MAPLE unifies them by replacing with a unique symbol, `rcv`. Similarly, MAPLE replaces the input argument with `arg0`, as shown in Column Symbolized.

MAPLE initializes each cluster with each generalized guard. In the following clustering process, MAPLE checks the equivalence of every pair of clusters and merges them with if the guards are logically equivalent, until no more clusters can be merged. At the end, we count the number of guard conditions in each cluster as frequency. Since the same logical predicate can be expressed in multiple ways in different projects, prior work on predicate mining checks syntactic similarity after applying rewriting heuristics [30], for example by converting both `arg0<arg1` and `arg1>arg0` to `arg0<arg1`. To overcome the limitation of relying on syntactic similarity only, MAPLE formalizes the equivalence of two guard conditions as a satisfiability problem:

$$p \Leftrightarrow q \text{ is valid iff. } \neg((\neg p \vee q) \wedge (p \vee \neg q)) \text{ is unsatisfiable.}$$

MAPLE uses a SMT solver, Z3 [23] to check the logical equivalence between two guards during the merging process. As Z3 only supports primitive types, MAPLE declares variables of unsupported data types as integer variables and substitute constants such as `null` with integers in Z3 queries. In addition, MAPLE substitutes API calls in a predicate to symbolic variables based on their return types. Compared with prior work, MAPLE is capable of proving the semantic equivalence of arbitrary predicates regardless of their syntactic similarity. For example, the symbolized guards of the first two examples in Table II are equivalent, even though they are expressed in different ways, `-1<arg0 && arg0<rcv.length()+1` and `0<=arg0 && arg0<=rcv.length()` respectively. Prior work [30] cannot reason about the equivalence between `-1<arg0` and `0<=arg0`. However, MAPLE groups these predicates into the same cluster using the integer theorem prover in Z3.

Similar to the subsequence mining in Phase 2, MAPLE also requires users to specify a minimum support threshold θ to filter infrequent guard conditions. The default threshold θ is set to 0.5 to mine a guard condition supported by at least half of code snippets using the given API. MAPLE then composes the desired API usage by augmenting the common subsequence with a common guard condition for each call. MAPLE enumerates all combinations, if a sequence pattern contains multiple API calls and each API call could be protected by multiple guard patterns. The composed API

usage patterns are ranked by the number of code examples that support each pattern in the corpora.

D. API Misuse Detection

MAPLE may learn patterns of no interest for two reasons. First, shorter patterns with fewer API calls are often supported by more examples. Therefore, MAPLE may rank incomplete patterns higher than complete ones. Second, if desirable API usage occurs infrequently on GitHub, MAPLE may not easily infer the correct pattern. For instance, `PrintWriter.close` should be called in `finally` to ensure that the buffered contents are written to the stream in case an exception occurs in the middle of execution. However, among all code snippets calling `PrintWriter.close` on GitHub, only 17% call `close` in `finally`. As a result, this pattern is ranked #14 in all inferred patterns of `PrintWriter.close`. Therefore, MAPLE allows users to inspect inferred patterns to reduce mis-identified API usage violations, before comparing them against code snippets in Stack Overflow posts.

When multiple patterns are inferred, MAPLE also allows users to annotate each pattern as either *alternative* or *required.* A code snippet should satisfy one of the *alternative* pattern, and must satisfy all *required* patterns. For example, programmers can check file existence before creating a new file by either calling `File.exists` or checking the return value of `File.createNewFile`, which are considered as alternative patterns. Programmers must handle potential `IOException` when reading from a stream and also close the stream at the end to avoid resource leaks, which are considered as required patterns.

Given a human-annotated pattern, MAPLE checks whether the structured call sequence of a Stack Overflow snippet is *subsumed* by the pattern. A structured call sequence s is subsumed by a pattern p, only if p is a subsequence of s and the guard condition of each API call in s implies the guard of the corresponding API call in p. During this subsumption checking process, the guard conditions in Stack Overflow code snippets are generalized in the same manner before checking logical implication using Z3.

IV. PATTERN MINING EVALUATION

This section investigates whether MAPLE is capable of mining API usage patterns *efficiently* and *accurately* on massive code corpora. Our study uses 30 Java and Android API methods selected from an existing API misuse benchmark, MUBENCH [22]. The benchmark includes API misuse from existing bug datasets [31]–[33], previous literature [3, 34], and API misuse reported by professional developers. We exclude APIs that have no references on Stack Overflow and those with project-specific misuse related to the logic of a client program. In Table III, API shows individual API methods in our study. API misuse description describes the particular misuse studied by MUBENCH, and API Usage Patterns shows the ground-truth patterns confirmed by MUBENCH and further augmented by the first author based on the corresponding JavaDoc documentation.

Setting	Precision (%)			Recall (%)			Rank
	Top 3	Top 5	Top 10	Top 3	Top 5	Top 10	
Small Corpus	49	49	49	42	46	53	6
Massive Corpora	79	80	79	85	91	94	3

TABLE IV: Pattern Mining Accuracy Comparison

MAPLE learns not only confirmed patterns in MUBENCH but also additional patterns consistent with JavaDoc documentation. For instance, `TypedArray` is allocated from a static pool to store the layout attributes, whenever a new application view is created in Android. The corresponding pattern in MUBENCH checks for missing exception handling, when retrieving attributes from `TypedArray` with invalid indices, which is confirmed on GitHub.[3] From another perspective, `TypedArray` should be recycled immediately to avoid resource leaks and GC overhead, as mentioned in the JavaDoc.[4] This pattern is supported by 2,206 code snippets in GitHub and inferred by MAPLE (ranked #1).

A. Scalability

MAPLE scans over 7 million GitHub projects and finds an average of 32,678 relevant code snippets for each API method, ranging from 376 to 294,569 snippets. This result indicates that massive code corpora can provide sufficient code snippets to learn patterns from. On average, MAPLE takes around 10 minutes to search over the massive corpora and find relevant GitHub code snippets for each API. Data dependence-based slicing does not impose much overhead during code search.

To demonstrate that MAPLE scales to a large number of code examples, we repeat existing GitHub snippets to create a dataset with 100k code snippets for each API method and run MAPLE on these datasets using the default setting (k=1, σ=0.5, θ=0.5). The experiments are run on a single machine with 2.93GHz dual core processor and 8GB DDR3 RAM. We run MAPLE on each dataset of each API five times and compute the average execution time. On average, MAPLE's pattern inference takes about 4 minutes using the default setting. The sensitivity analysis using different dependency bounds and pattern mining thresholds is detailed in Section VI.

B. Accuracy

Table IV shows the precision and recall of inferred patterns with respect to the ground-truth patterns in Column API Usage patterns in Table III. MAPLE is capable of learning expected patterns for all 30 APIs from massive code corpora with 80% precision and 91% recall when considering top 5 patterns for each API method. As we inspect top 10 patterns for each API method, the recall increases gradually to 94%, while the precision does not vary much. To demonstrate the benefit of mining massive code corpora, we apply MAPLE to a small code corpus that contains 7,899 randomly selected GitHub projects only. MAPLE learns expected patterns for 19

[3]https://github.com/chrisjenx/Calligraphy/issues/41

[4]https://developer.android.com/reference/android/content/res/TypedArray.html

ID	API Name	API Misuse Description	API Usage Patterns	Severity	Total	Unreliable
1	File.createNewFile	The file may exist before creating it.	createNewFile(0)@!rcv.exists() ✓ mkdirs(0)@true; createNewFile(0)@true ✓ createNewFile(0)@true; if {; } ∗	other	848	337
2	Iterator.next	`NoSuchElementException` is thrown if no more elements exist.	next(0)@rcv.hasNext() ✓	crash	4536	447
3	HashMap.get	A null pointer is returned if no such key exists.	get(1)@true; if{; } ✓ get(1)@rcv.containsKey(arg0) ∗ get(1)@rcv!=null ∗	crash	970	733
4	File.mkdirs	call `mkdir`, which does not create nonexistent parent directories.	mkdirs(0)@true ✓	crash	797	4
5	Cipher.init	`init` may throw `InvalidKeyException`.	try {; init(2)@true; }; catch {; } ✓	crash	329	113
6	InputStream.read	`InputStream` is not closed.	read(1)@true; close(0)@true ✓ try{; read(1)@true; }; catch {; } ∗	res leak / crash	335	198
7	StringTokenizer.nextToken	`NoSuchElementException` is thrown if no more tokens exist.	nextToken(0)@rcv.hasMoreTokens() ✓	crash	729	238
8	DataOutputStream.write	`DataOutputStream` is not closed.	write(1)@true; close(0)@true ✓	res leak	112	62
9	SortedMap.firstKey	`NoSuchElementException` is thrown if the map is empty.	firstKey(0)@!(rcv==null||rcv.isEmpty()) ✓ firstKey(0)@rcv.size()>0 ∗	crash	22	21
10	ApplicationInfo.loadIcon	`OutOfMemoryError` is thrown when icons are large.	try {; loadIcon(1)@true; }; catch {; } ✓	crash	16	16
11	TypedArray.getString	`IndexOutOfBoundsException` is thrown if the index is not valid.	try {; getString(1)@true }; catch {; } ✓ getString(1)@true; recycle(0)@true ∗	crash	15	15
12	RandomFileAccess.close	`close` can be skipped.	finally {; close(0)@true; } ✓ try {; close(0)@true; }; catch {; } ∗ if {; close(0)@rcv!=null; } ∗	res leak / crash	141	131
13	RandomFileAccess.read	`RandomFileAccess` is not closed after reading.	read(1)@true; close(0)@true ✓ try {; read(1)@true; }; catch {; } ∗ read(1)@true; if {; } ∗	res leak / crash	34	22
14	RandomFileAccess.write	`RandomFileAccess` is not closed after writing.	write(1)@true; close(0)@true ✓ try {; write(1)@true; }; catch {; } ∗	res leak / crash	47	21
15	Activity.setContentView	Throw exceptions if calling `setContentView` before `super.onCreate`	super.onCreate(1)@true; setContentView(1)@true ✓	crash	4425	35
16	Activity.findViewById	Return `null` if no such view exists.	findViewById(1)@true; if {; }✓	crash	27	24
17	PrintWriter.close	`close` can be skipped.	finally {; close(0)@true; } ✓ try {; close(0)@true; }; catch {; } ∗ if {; close(0)@rcv!=null; } ∗	res leak / crash	745	536
18	PrintWriter.write	`PrinterWriter` is not closed after writing.	write(1)@true; close(0)@true ✓ try {; write(1)@true; }; catch {; } ∗ write(1)@true; flush(0)@true ∗	res leak / crash	215	117
19	new FileInputStream	Throw `FileNotFoundException` if the file does not exist.	new FileInputStream(1)@arg0.exists() ✓ try {; new FileInputStream(1)@true; }; catch {; } ∗	crash	6101	2788
20	ByteBuffer.put ByteBuffer.getInt	Previous data is not read properly without flipping the buffer.	put(1)@true; flip(0)@true; getInt(0)@true ✓ put(1)@true; rewind(0)@true; getInt(0)@true ∗ put(1)@true; position(1)@true; getInt(0)@true ∗	crash	11	4
21	FileChannel.write	The written content is not flushed before the next read	write(1)@true; close(0)@true ✓ try {; write(1)@true; }; catch {; } ∗	res leak / crash	38	21
22	ArrayList.get	Throw `IndexOutOfBoundsException` if the index exceeds the limit.	get(1)@arg0<rcv.size() ✓	crash	2150	1328
23	JFrame.setPreferredSize	The preferred size is never reflected without calling `pack`.	setPreferredSize(1)@true; pack(0)@true ✓ setPreferredSize(1)@true; setVisible(1)@true ∗	other	138	19
24	JsonElement.getAsString	Throw `ClassCastException` if the element is not valid.	getAsString(0)@rcv.isJsonPrimitive() ∗ getAsString(0)@rcv!=null ∗	crash	58	58
25	ProgressDialog.dismiss	Throw `IllegalArgumentException` if the dialog is not showing.	dismiss(0)@rcv.isShowing() ✓ dismiss(0)@rcv!=null ∗	crash	193	179
26	SQLiteDatabase.query	The cursor returned by `query` is not closed.	query(7)@true; close(0)@true ✓ query(7)@true; if {; } ∗	crash	48	33
27	String.getBytes	String is not encoded correctly without specifying the desired charset.	getBytes(1)@true ✓	other	2416	1415
28	String.getBytes Mac.doFinal	MAC does not operate properly with incorrectly encoded bytes.	getBytes(1)@true; doFinal(1)@true ✓ try {; getBytes(0)@true; doFinal(1)@true; }; catch {; } ∗	other	30	19
29	Mac.doFinal, new String	The MAC result is not converted correctly without specifying the charset.	doFinal(1)@true; encode(1)@true; new String(1)@true ✓ doFinal(1)@true; encodeBase64(1)@true; new String(1)@true∗ doFinal(1)@true; new String(2)@true ∗ try {, doFinal(1)@true, new String(1)@true, }; catch {; } ∗ init(2)@true, doFinal(1)@true, new String(1)@true ∗	other	282	249
30	new SimpleDateFormat	`Date` is not displayed correctly without specifying the locale.	new SimpleDateFormat(2)@true ✓	other	8163	7149

TABLE III: Pattern mining and API misuse detection benchmark. Patterns annotated with ✓ are confirmed by prior work [22] while patterns annotated with ∗ are validated by the authors based on JavaDoc.

of 30 APIs from the small corpus with 49% precision and 46% recall in top 5 patterns. The average rank of expected patterns is 6 when using the small corpus vs. 3 when using the massive corpora, indicating using massive corpora does indeed provide the benefit of placing good patterns on the top. MAPLE learns patterns with low accuracy and ranking in the small corpus for three reasons. In 3 cases, MAPLE does not find any relevant snippets in the small corpus. In 6 cases, MAPLE finds several snippets but none of them contain correct API usage. In 2 cases, MAPLE finds some snippets but the correct usage occurs infrequently. Unless there is an efficient way of creating a corpus of high-quality code examples, mining massive corpora rather than a pre-defined corpus is necessary for inferring correct API usage patterns for arbitrary APIs.

V. API MISUSE STUDY ON STACK OVERFLOW

To assess the reliability of code snippets included in Stack Overflow posts, we use the set of patterns inferred by MAPLE and further validated by the first author to detect API misuse in Stack Overflow posts.

A. Data Collection

We collect all posts relevant to the 30 APIs in Table III from the Stack Overflow data dump.[5] We first extract code snippets in the markdown `<code>` from SO posts with the `Java` tag. We only include code snippets in answer posts, since code in question posts is buggy and rarely used as examples. We also gather additional information associated with each post, including view counts, scores (i.e., upvotes minus downvotes), and whether a post is accepted as the correct answer.

For each API in the benchmark, we parse Stack Overflow snippets using a customized Java parser for partial programs and only retain those examples that call the API method under focus. Our parser wraps an arbitrary snippet with mocked class and method headers as needed and resolves types and API calls with an API oracle extracted from JDK and Android SDK [35]. Code examples that call overridden APIs or ambiguous APIs (i.e., APIs with the same name but from different Java classes) are filtered by checking the argument types and receiver types respectively. In total, we find 31,801 SO posts with code examples for the 30 APIs in our benchmark. Each post has 10,890 view counts on average. Column Total in Table III shows the number of posts for individual APIs. Please note that one post may reference multiple APIs in our benchmark.

B. Is API Misuse Prevalent on Stack Overflow?

We detect API misuse in 16,098 (51%) out of 31,801 Stack Overflow posts in our study. Column Unreliable shows the number of posts with API misuse for individual APIs. Figure 4 compares APIs from different domains. Many examples misuse Android APIs due to undocumented behavior or contracts [36]. Cryptography APIs are also often misused on Stack Overflow, which is consistent with the previous observation that programmers often struggle with cryptography APIs [34, 37]. Among posts with potential API misuses reported by MAPLE, 37% include the types of API violation that could throw unexpected exceptions, 7% could lead to handle leaks in operating systems, 12% could lead to corrupt data (e.g., incorrectly encoded bytes), and 44% could potentially cause representation issues (e.g., ill-formed dates), according to our inspection of the differences between the embedded code snippets and the desired API usage. To support our argument that these API usage violations could indeed induce real-world bugs, we publish bug reports and commits that report the corresponding API usage violations in GitHub and SourceForge.[6]

We fully acknowledge that not all API usage violations could lead to bugs in client applications. There are two reasons. First, a client program can be resilient to an API usage violation. For example, if `NoSuchElementException` is caught and handled properly by the caller, `Iterator.next` (#2 in Table III) will not crash a program, even though no more elements exist. Second, an API is used in a specific context where the error-inducing condition never occurs. For example, if `File.createNewFile` (#1 in Table III) creates a file that is guaranteed non-existent in a target client program, this operation will always succeed. Please note that reasoning about such cases requires deep understanding about the semantics of a client program. To more accurately assess the runtime impact of code examples, this study can be extended by systematically integrating SO code snippets to real-world client applications and running them via regression testing.

[5]https://archive.org/details/stackexchange, accessed on Oct 17, 2016.

[6]The link to our repository is removed for double-blind review. We will release the tool and dataset, once the paper is accepted.

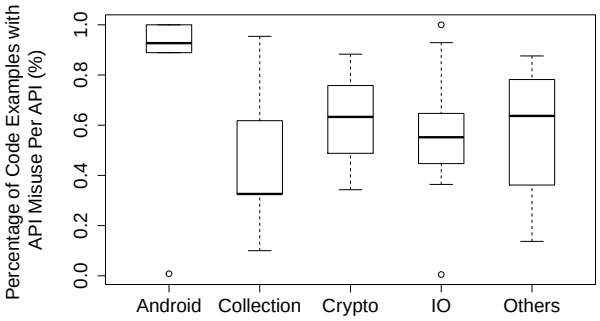


Fig. 4: API Misuse Comparison between Different Domains

C. What are the characteristics of API misuse?

We classify the detected API misuses into four categories based on the required edits to correct an API misuse to follow the corresponding usage pattern.

Missing Control Constructs. Many APIs should be used in a specific control-flow context to avoid unexpected behavior. This type of API misuses can be further split based on the type of missing control constructs.

Missing exception handling. If an API may throw an exception, the thrown exception should either be caught and handled a `try-catch` block or be declared in the method header. In total, we find 2,985 code examples that do not handle exceptions properly.

Missing finally. Clean-up APIs such as `close` should be invoked in a finally block in case an exception occurs before invoking those APIs. For instance, `br.close` in Figure 1b can be skipped if `FileNotFoundException` is thrown by `new FileReader`, leading to unmanaged file resources. Most examples (83%) of clean-up APIs (#12 and 17 in Table III) do not include a finally block.

Missing if checks. Some APIs may return erroneous values such as null pointers, which must be checked properly to avoid crashing the succeeding execution. Invoking `get` will return `null`, if the requested key does not exist in the map (line 10).[7] Since `null` cannot be returned as a `double` value, the example will throw `NullPointerException` at line 10. 1,077 code examples do not check the return value of an API call properly on Stack Overflow.

```
1 public class Loan {
2   private static HashMap<Integer,Double> rate = new
        HashMap<Integer,Double>();
```

[7]http://stackoverflow.com/questions/29689729

```
3   Loan()
4   {
5     rate.put(15, 3.25);
6     rate.put(30, 3.95);
7   }
8   public double getRate(int year) {
9     //Now you can get the desired rate
10    return rate.get(year);
11  }
12 }
```

Missing or Incorrect Order of API calls. In certain cases, multiple APIs should be called together in a specific order to achieve desired functionality. Missing or incorrect order of such API calls can lead to unexpected behavior. For example, developers must call `flip`, `rewind`, or `position` to reset the internal cursor of `ByteBuffer` back to the previous position to read the buffered data properly. However, the following example will throw `BufferUnderflowException` because the cursor has already reached the upper bound of the buffer after the write operation at line 2.[8] Without resetting the cursor, the next read operation at line 3 will start reading from the upper bound, which is prohibited. We find 580 Stack Overflow posts that miss at least one critical API call on Stack Overflow.

```
1 ByteBuffer bb = ByteBuffer.allocate(4);
2 bb.put(newArgb);
3 int i = bb.getInt();
```

As another example, `super.onCreate` must be invoked first to initialize the Android app correctly. However, 35 posts include code snippets invoking `Activity.setContentView` (#15 in Table III), leading to `NullPointerException` because the layout objects are not initialized yet. We find 129 posts that call APIs in an incorrect order.

Incorrect Guard Conditions. Many APIs should be invoked under the correct guard condition to avoid runtime exceptions. For instance, programmers should check whether a sorted map is empty with a guard like `map.size()>0` or `!map.isEmpty()` before calling `firstKey` (API#9) on the map. However, the following example calls `firstKey` on an empty map without a guard, leading to `NoSuchElementException`.[9] Surprisingly, this example is accepted as the correct answer and also upvoted by six other developers on Stack Overflow. We find 5,488 posts calling APIs with incorrect guard conditions.

```
1 TreeMap map = new TreeMap();
2 //OR SortedMap map = new TreeMap()
3 map.firstKey();
```

D. Are recognized code examples more reliable?

67% of the studied posts are either accepted as correct answers or have more upvotes than downvotes. Such recognized posts are often considered to have higher quality than unrecognized ones. Previous study has shown that programmers often pay more attention to accepted answers [38]. However, we observe that recognized posts have as many API usage violations as unrecognized ones. Overall, 51% of recognized posts have API misuse, while 50% of unrecognized posts have API misuse. Figure 5 shows that the median percentage of posts with API misuse does not have much difference between recognized posts and unrecognized posts.

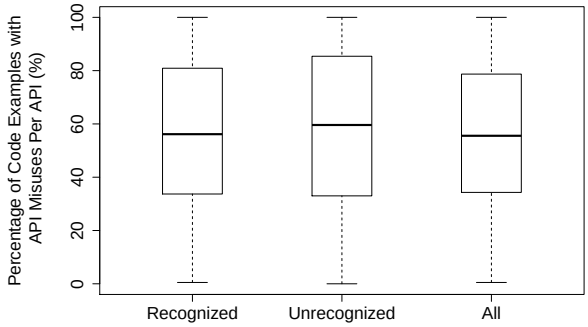


Fig. 5: API Misuse Comparison between Recognized and Unrecognized Examples on Stack Overflow

Each post on Stack Overflow is assigned a score (i.e., upvotes minus downvotes). The average score of recognized posts in our dataset is 5, indicating endorsement from 5 programmers on Stack Overflow. We investigate the correlation between the reliability of recognized posts and their scores on Stack Overflow. We perform a linear regression on the score and the percentage of unreliable examples, as shown by the red line in Figure 6. However, we do not observe a strong positive or negative correlation between the score of a post and its reliability in terms of API misuse.

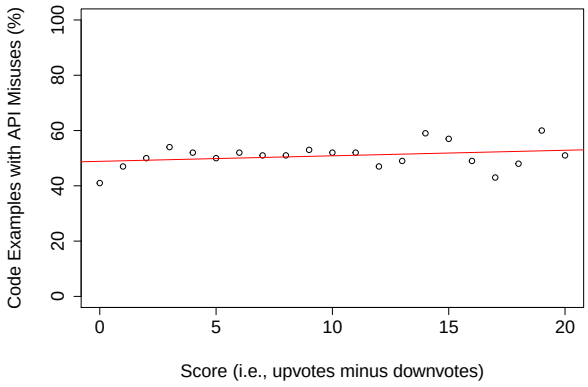


Fig. 6: Percentage of Recognized Examples with API Misuse from Score 0 to 20.

E. Manual Inspection

To check whether Stack Overflow posts with potential API misuse reported by MAPLE indeed suggest undesirable API usage, the first and the third authors randomly select up to 20 Stack Overflow posts for each API (574 in total) and manually check the code snippets included in those posts. To reduce subjectivity, the two authors inspect these posts independently and then discuss any disagreements. This process takes around 35 hours per person. The authors confirm API misuse in 288

[8]http://stackoverflow.com/questions/12100651
[9]http://stackoverflow.com/questions/21983867

stack overflow posts (50%). The authors still disagree on 3 posts after discussion, since the semantics of the code snippets in those 3 posts are ambiguous. Within this sampled data set of 574 posts, MAPLE flags posts with undesirable API usage in 295 posts with 85% precision and 96% recall. Please note that this accuracy assessment is about flagging undesirable or incorrect API usage in Stack Overflow *posts*, not about assessing the accuracy of inferred API usage *patterns*.

a) False Negative Posts: MAPLE does not flag API misuse in 13 posts, due to the abstraction in the structured call sequence model. First, since our model abstracts away the receivers of API calls, we cannot easily tell whether two APIs are called on the same object or not during API misuse detection. In 11 posts, expected APIs are called in a correct order but are called on irrelevant objects. Second, our model also abstracts away details in control constructs such as if conditions and caught exceptions. In 2 posts, an `if` statement after an API call does not check the return value of the API call, and a `catch` clause does not catch the expected exception thrown by an API.

b) False Positive Posts: MAPLE mistakenly detects API misuse in 42 posts for three reasons. First, our approach extracts API call sequences from each single method without considering API calls between multiple methods. In 3 posts, an expected API call or its guard condition occurs in the parent caller of the method calling the API of interest. For instance, the following example reads an input stream in `readBoolean` but then closes it in `main`.[10]

```
1 public static void main() {
2   ...
3   readBoolean(in);
4   in.close();
5 }
6 private static void readBoolean(InputStream in) {
7   in.read();
8 }
```

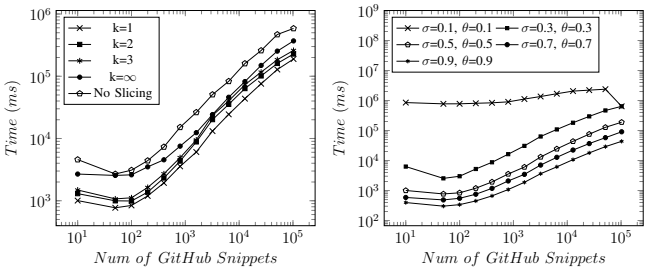
Second, MAPLE checks whether a code snippet follows a pattern via sequence comparison, which is not sufficient in 15 posts. For instance, the following post calls `HashMap.get` (line 6) without checking whether the key exists.[11] However, the key is guaranteed to exist in the `HashMap` object, because the key is obtained from the key set of the same `HashMap` object via an iterator (line 2). Such cases can be addressed by more advanced analysis by accounting for aliasing and full symbolic execution.

```
1 HashMap characters = ...
2 Set keys = characters.keySet();
3 Iterator it = keys.iterator();
4 while(it.hasNext()) {
5   String key = (String) it.next();
6   String replacement = (String) characters.get(key);
7   ...
8 }
```

Third, in 21 posts, Stack Overflow programmers comment informally that the code example can be improved by adding an extra API call or a guard condition in surrounding natural language texts. However, MAPLE does not check the surrounding text and therefore may still report API usage violations.

[10]http://stackoverflow.com/questions/9349545

[11]http://stackoverflow.com/questions/222361



(a) Dependency Analysis Bound (b) Minimum Support Threshold

Fig. 7: Pattern mining performance.

Setting	Precision (%)			Recall (%)			Rank
	Top 3	Top 5	Top 10	Top 3	Top 5	Top 10	
k = 1	79	80	79	85	91	94	3
k = 2	77	79	79	84	92	95	4
k = 3	77	80	78	84	91	94	4
k = ∞	71	74	73	77	91	94	4
No Slicing	65	65	67	73	81	89	9

TABLE V: Pattern mining accuracy using different k bounds.

VI. DISCUSSION

Sensitivity Analysis. Figure 7a shows the pattern inference with the default thresholds (σ=0.5, θ=0.5) and with different hop bounds for k. We run each experiment five times and compute the average execution time. We kill a mining process, if it does not terminate in 2 hours. Setting k to ∞ retains all dependent API calls in a sliced call sequence, while setting k to 1 retains only immediately dependent statements. Setting k to 1 can achieve 3.3X speed up compared with setting k to ∞, since it creates shorter API call sequences by removing transitively dependent API calls. MAPLE runs up to 4.6X slower when not removing irrelevant API calls (no slicing). Figure 7b shows the result of setting k to 1 while varying minimum support thresholds for sequence mining and guard condition mining. MAPLE slows down significantly as the thresholds go below 0.5. MAPLE takes more than 2 hours to mine patterns for three API methods, when setting the minimum thresholds to 0.1, since MAPLE learns too many candidate subsequences above the thresholds. On average, MAPLE's running time is within 10 minutes, with thresholds above 0.3.

Table V shows the pattern mining accuracy using different k bounds. Even though bounding dependency analysis with lower bounds may lead to incomplete sequences with fewer API calls, varying k does not affect accuracy much. However, compared with unbounded analysis, filtering out transitively dependent API calls can improve precision and recall slightly. This is because long API call sequences may introduce additional patterns of no interest. Furthermore, mining call sequences without removing any irrelevant calls through slicing

may add more noise, which may degrade precision and recall by 14% and 12% respectively in top 3 patterns.

Threats to Validity. Our benchmark only includes 30 Java and Android APIs from an existing API-misuse benchmark. Our study may not generalize to APIs that programmers are less likely to make mistakes on. Our study scope is limited to code snippets found on Stack Overflow. Other types of online resources such as programming blogs and other Q&A forums may have better curated examples. MAPLE may overlook or mis-identify API misuses due to the limitations discussed in Section V-E. The two authors have manually inspected randomly selected Stack Overflow posts and measured whether the posts flagged by MAPLE indeed include undesirable or incorrect API usage, reporting 93% precision and 96% recall respectively. However, we acknowledge that we sampled only 574 out of 31,801 posts.

VII. Related Work

Quality Assessment of Online Code Examples. Prior work has investigated the quality of online code examples from different perspectives. Nasehi et al. [38] find that code examples and detailed, step-by-step explanations are the two key elements in highly voted Stack Overflow posts. Dagenais and Robillard [10] find that 89% of API names in code snippets from online forums are ambiguous and cannot be easily resolved due to the incompleteness of these snippets. Subramanian et al. [8] 66% of 39k Stack Overflow snippets are free-standing statements without class or method declarations. Yang et al. [9] observe a similar result—3.89% of 91k Java code snippets on Stack Overflow are parsable and only 1% are compilable. Zhou et al. [11] find that 86 of 200 accepted posts on Stack Overflow use deprecated APIs but only 3 of them are reported by other programmers. However, none of these studies have investigated the reliability of online code examples in the sense that following these code examples verbatim may encourage potential API misuse and consequently lead to unexpected behavior such as program crashes in client programs. To fill the gap, our work presents the first empirical study that flags API misuses on Stack Overflow using API usage patterns learned from massive code corpora.

API Usage Mining. There is a large body of literature in API usage mining [12]–[20, 27, 39]. GrouMiner [12] models programs as graphs and performs frequent subgraph mining to find API usage patterns. However, their evaluation shows that GrouMiner takes 16 mins to mine a single project on average, which may not scale to massive code corpora with millions of projects. Buse et al. [17] also model programs as graphs but cluster them using the k-medoid algorithm [40]. Gruska et al. [39] extract call sequences from programs and perform formal concept analysis [41] to infer pairwise temporal properties of API calls. Similar to Gruska et al., there are many other specification mining techniques dedicated to inferring temporal properties of API calls [42]–[47]. UP-Miner [18] mines frequent sequence patterns but does not retain control constructs and guard conditions in API usage patterns. Several techniques [13, 15, 16] model programs as item sets and infer pairwise programming rules using frequent itemset mining [48], which does not consider temporal ordering or guard conditions of API calls.

MAPLE differs from existing pattern mining techniques in two perspectives. First, MAPLE mines from massive code corpora with millions of GitHub projects. Gruska et al. mine from 6,000 Linux projects, which is the largest code corpus we are aware of. However, it is still several orders of magnitude smaller than MAPLE. Second, MAPLE mines not only API call ordering but also API call guard conditions by using a novel predicate mining technique. To our best knowledge, Ramanathan et al. [49] and Nguyen et al. [30] are the only two predicate mining techniques. Ramanathan et al. apply inter-procedure data-flow analysis to collect all predicates related to a callsite and then use frequent itemset mining to find common predicates. Unlike MAPLE, Ramanathan et al. only mine a single project and cannot handle semantically equivalent predicates in different forms. Nguyen et al. improve upon Ramanathan et al. by normalizing predicates using several rewriting heuristics. Unlike these techniques, MAPLE formalizes the predicate equivalence problem as a satisfiability problem and leverages a SMT solver to group logically equivalent predicates during guard mining.

VIII. Conclusion

Programmers often resort to code snippets in online Q&A forums such as Stack Overflow to learn about how to use APIs correctly during software development. However, the reliability of code snippets in Stack Overflow posts is under-investigated. In this paper, we present an approach called MAPLE that compares online code snippets against API usage mined from over 7 million Github projects and reports potential API usage violations in those snippets. Our study shows that more than half of SO posts in our study contain potential API usage violations that may produce the symptoms of crash and resource leaks when used verbatim in client applications. This finding demonstrates the prevalence and severity of API misuse in code snippets on Stack Overflow.

The API usage mining technique implemented in MAPLE combines efficient call sequence mining with guard condition mining to retain both API call ordering and guard conditions to protect individual API calls. MAPLE scales to millions of projects, expanding the capability of API usage mining to an unprecedented scale and does not sacrifice the fidelity and expressiveness of its API usage representation by leveraging control and data flow analysis. Our work provides a foundation for a new human-in-the-loop approach to enrich and enhance code snippets included in an online, collaborative Q&A forum by contrasting them against frequent usage evidenced by millions of Github projects.

References

[1] M. Umarji, S. E. Sim, and C. Lopes, "Archetypal internet-scale source code searching," in *IFIP International Conference on Open Source Systems*. Springer, 2008, pp. 257–263.

[2] C. Sadowski, K. T. Stolee, and S. Elbaum, "How developers search for code: a case study," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 2015, pp. 191–201.

[3] Q. Gao, H. Zhang, J. Wang, Y. Xiong, L. Zhang, and H. Mei, "Fixing recurring crash bugs via analyzing q&a sites (t)," in *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, 2015, pp. 307–318.

[4] F. Chen and S. Kim, "Crowd debugging," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 2015, pp. 320–332.

[5] C. Treude and M. P. Robillard, "Augmenting api documentation with insights from stack overflow," in *Proceedings of the 38th International Conference on Software Engineering*. ACM, 2016, pp. 392–403.

[6] E. Wong, J. Yang, and L. Tan, "Autocomment: Mining question and answer sites for automatic comment generation," in *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*. IEEE, 2013, pp. 562–567.

[7] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Summarizing source code using a neural attention model," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*, vol. 1, pp. 2073–2083.

[8] S. Subramanian and R. Holmes, "Making sense of online code snippets," in *Proceedings of the 10th Working Conference on Mining Software Repositories*. IEEE Press, 2013, pp. 85–88.

[9] D. Yang, A. Hussain, and C. V. Lopes, "From query to usable code: an analysis of stack overflow code snippets," in *Proceedings of the 13th International Workshop on Mining Software Repositories*. ACM, 2016, pp. 391–402.

[10] B. Dagenais and M. P. Robillard, "Recovering traceability links between an api and its learning resources," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 47–57.

[11] J. Zhou and R. J. Walker, "Api deprecation: a retrospective analysis and detection method for code examples on the web," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 266–277.

[12] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen, "Graph-based mining of multiple object usage patterns," in *ESEC/FSE '09: Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. New York, NY, USA: ACM, 2009, pp. 383–392.

[13] 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.

[14] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman, "Jungloid mining: helping to navigate the api jungle," in *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*. New York, NY, USA: ACM, 2005, pp. 48–61.

[15] 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.

[16] S. Thummalapenta and T. Xie, "Alattin: mining alternative patterns for defect detection," *Automated Software Engineering*, vol. 18, no. 3, p. 293, 2011.

[17] R. P. Buse and W. Weimer, "Synthesizing api usage examples," in *Software Engineering (ICSE), 2012 34th International Conference on*. IEEE, 2012, pp. 782–792.

[18] J. Wang, Y. Dang, H. Zhang, K. Chen, T. Xie, and D. Zhang, "Mining succinct and high-coverage api usage patterns from source code," in *Proceedings of the 10th Working Conference on Mining Software Repositories*. IEEE Press, 2013, pp. 319–328.

[19] M. Raghothaman, Y. Wei, and Y. Hamadi, "Swim: synthesizing what i mean: code search and idiomatic snippet synthesis," in *Proceedings of the 38th International Conference on Software Engineering*. ACM, 2016, pp. 357–367.

[20] X. Gu, H. Zhang, D. Zhang, and S. Kim, "Deep api learning," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 631–642.

[21] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen, "Boa: A language and infrastructure for analyzing ultra-large-scale software repositories," in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 422–431.

[22] S. Amani, S. Nadi, H. A. Nguyen, T. N. Nguyen, and M. Mezini, "Mubench: a benchmark for api-misuse detectors," in *Proceedings of the 13th International Conference on Mining Software Repositories*. ACM, 2016, pp. 464–467.

[23] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[24] F. E. Allen, "Control flow analysis," in *ACM Sigplan Notices*, vol. 5, no. 7. ACM, 1970, pp. 1–19.

[25] M. Weiser, "Program slicing," in *Proceedings of the 5th international conference on Software engineering*. IEEE Press, 1981, pp. 439–449.

[26] J. Wang, J. Han, and C. Li, "Frequent closed sequence mining without candidate maintenance," *IEEE Transactions on Knowledge and Data Engineering*, vol. 19, no. 8, 2007.

[27] S. Thummalapenta and T. Xie, "Parseweb: a programmer assistant for reusing open source code on the web," in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. ACM, 2007, pp. 204–213.

[28] ——, "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.

[29] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei, "Mapo: Mining and recommending api usage patterns," in *European Conference on Object-Oriented Programming*. Springer, 2009, pp. 318–343.

[30] H. A. Nguyen, R. Dyer, T. N. Nguyen, and H. Rajan, "Mining preconditions of apis in large-scale code corpus," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 166–177.

[31] K. Herzig, S. Just, and A. Zeller, "It's not a bug, it's a feature: how misclassification impacts bug prediction," in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 392–401.

[32] 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*. ACM, 2014, pp. 437–440.

[33] V. Dallmeier and T. Zimmermann, "Extraction of bug localization benchmarks from history," in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. ACM, 2007, pp. 433–436.

[34] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel, "An empirical study of cryptographic misuse in android applications," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 2013, pp. 73–84.

[35] S. Subramanian, L. Inozemtseva, and R. Holmes, "Live api documentation," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 643–652.

[36] M. Kechagia and D. Spinellis, "Undocumented and unchecked: exceptions that spell trouble," in *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM, 2014, pp. 312–315.

[37] S. Nadi, S. Krüger, M. Mezini, and E. Bodden, "Jumping through hoops: why do java developers struggle with cryptography apis?" in *Proceedings of the 38th International Conference on Software Engineering*. ACM, 2016, pp. 935–946.

[38] S. M. Nasehi, J. Sillito, F. Maurer, and C. Burns, "What makes a good code example?: A study of programming q&a in stackoverflow," in *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*. IEEE, 2012, pp. 25–34.

[39] N. Gruska, A. Wasylkowski, and A. Zeller, "Learning from 6,000 projects: lightweight cross-project anomaly detection," in *Proceedings of the 19th international symposium on Software testing and analysis*. ACM, 2010, pp. 119–130.

[40] L. Kaufman and P. J. Rousseeuw, *Finding groups in data: an introduction to cluster analysis*. John Wiley & Sons, 2009, vol. 344.

[41] B. Ganter and R. Wille, *Formal concept analysis: mathematical foundations*. Springer Science & Business Media, 2012.

[42] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," *IEEE Transactions on Software Engineering*, vol. 27, no. 2, pp. 99–123, 2001.

[43] G. Ammons, R. Bodik, and J. R. Larus, "Mining specifications," in *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. New York, NY, USA: ACM Press, 2002, pp. 4–16.

[44] 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.

[45] M. Pradel and T. R. Gross, "Automatic generation of object usage specifications from large method traces," in *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 2009, pp. 371–382.

[46] M. Gabel and Z. Su, "Online inference and enforcement of temporal properties," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. ACM, 2010, pp. 15–24.

[47] M. Pradel, C. Jaspan, J. Aldrich, and T. R. Gross, "Statically checking api protocol conformance with mined multi-object specifications," in *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 2012, pp. 925–935.

[48] G. Grahne and J. Zhu, "Efficiently using prefix-trees in mining frequent itemsets." in *FIMI*, vol. 90, 2003.

[49] 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.