Are Code Examples on an Online Q&A Forum Reliable?

A Study of API Misuse on Stack Overflow

ABSTRACT

Programmers often consult an online Q&A forum such as Stack Overflow to learn new APIs. This paper presents an empirical study on the prevalence and severity of API misuse on Stack Overflow. To reduce manual assessment effort, we design Maple, an API usage mining approach that extracts patterns from over 380K Java repositories on GitHub and subsequently reports potential API usage violations in Stack Overflow posts. We analyze 217,818 Stack Overflow posts using Maple and find that around 31% of them have potential API usage violations that may produce the symptoms such as 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*. Even the posts that are accepted as correct answers or upvoted by other programmers are not necessarily more reliable than other posts in terms of API misuse. This study result calls for a new human-in-the-loop approach to augment Stack Overflow code snippets and help the user consider better or alternative API usage.

CCS CONCEPTS

• **General and reference** → *Empirical studies*; • **Software and its engineering** → *Software reliability*; *Collaboration in software development*;

KEYWORDS

online Q&A forum, API usage pattern, code example comprehension

ACM Reference Format:
. 2018. Are Code Examples on an Online Q&A Forum Reliable?. In *Proceedings of International Conference on Software Engineering, Gothenburg, Sweden, May 2018 (ICSE 2018),* 11 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Programmers often resort to online code examples during software development [24, 32]. A case study at Google shows that developers issue an average of 12 code search queries per weekday [24]. Online code examples have also been utilized to facilitate other development activities, e.g, bug fixing [13], debugging [5], and API documentation [16, 31, 38].

ICSE 2018, May 2018, Gothenburg, Sweden

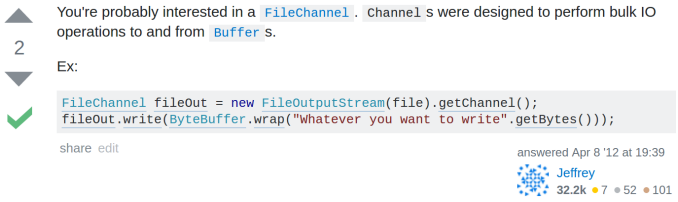
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Previous studies have analyzed the quality of code examples appearing in Q&A forums from different perspectives. Both Subramanian et al. [25] and Yang et al. [39] show that the majority of code examples on Stack Overflow are free-standing program statements that cannot be accepted by compilers. Dagenais and Robillard find that 89% of APIs mentioned in online forums cannot be easily resolved due to the ambiguity of API names [6]. Zhou et al. 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 [41]. None of these studies have investigated the reliability of online code examples in the sense that a code example may lead to API usage violations. Currently, it is left to the user to assess a given code example and enhance it as necessary during integration to a target application. There is no tool support to help developers easily recognize and repair unreliable code examples in online Q&A forums by leveraging code mining.

This paper aims to assess the reliability of Stack Overflow posts by contrasting code examples against desirable API usage patterns mined from GitHub. Our hypothesis is that commonly recurring API usage from massive corpora may represent a desirable pattern that a programmer can use to assess or enhance code examples on Stack Overflow. In particular, quantifying how many GitHub snippets are similar (or related but not similar) to the given example can provide confidence to a programmer about whether to trust the online code example as is. Therefore, we design API usage mining technology, Maple that scales to massive corpora without sacrificing the *fidelity* and *expressiveness* of the API usage representation. In particular, we use control and data flow analysis on top of an ultra-large-scale software mining infrastructure [8, 33], so that program slicing can be used to ignore irrelevant API calls. We combine frequent subsequence mining and SMT-based guard condition mining to retain both the temporal ordering of related API calls and appropriate guard conditions for each API call.

Maple efficiently searches over 380K GitHub repositories and retrieves an average of 55144 code snippets for a given API within 10 min. In terms of our study scope, we target one hundred Java and Android APIs that are popular and have corresponding posts in Stack Overflow. We then inspect all patterns inferred by Maple, create a data set of 180 desirable API usage patterns for the 100 APIs, and study the extent of API misuse in Stack Overflow posts.

Out of 217,818 SO posts relevant to our data set, 31% contain potential API misuse that could produce symptoms such as program crashes, resource leaks, and incomplete actions. Such API misuse is caused by three main reasons—*missing control constructs*, *missing or incorrect order of API calls*, and *incorrect guard conditions*. Database, IO, and networking APIs are often misused in Stack Overflow posts, since they often require observing temporal order between multiple calls and complex exception handling logic. 29% of *recognized posts*—those accepted as correct answers or endorsed by other programmers—have potential API usage violations, while 34% unrecognized posts have API usage violations. We do not observe



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

(b) An example that misses exception handling

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

a strong positive or negative correlation between the vote scores (i.e., upvotes minus downvotes) and the percentage 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 other posts. 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. As future work, we sketch a proposed Chrome extension that suggests better or alternative API usage for a given Stack Overflow code snippet, along with supporting concrete examples mined from GitHub that substantiate desirable or alternative API usage.

2 MOTIVATING EXAMPLES

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

Stack Overflow posts often provide a starting point for API investigation and thus do not need to disclose all details of API usage or include compilable, runnable code. However, if the user puts too much trust on the given example as is, she may inadvertently follow less than ideal API usage.

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 [30]. 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`. Calling `write` could 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.

As a novice programmer, Alice may not easily recognize the potential limitation of the given SO example. In this case, our approach Maple infers desirable API usage by scanning over 380K GitHub repositories and by finding 2230 Java methods that also call `FileChannel.write`. This mining process finds two common usage patterns. The mostly frequent usage supported by 1829 code snippets on GitHub indicates that a method call to `write()` must be contained inside a `try` and `catch` block. Another frequent usage supported by 1267 code snippets on GitHub indicates that `write` must be followed by `close`. By comparing code snippets in Figures 1a and 1b against these two API usage patterns, Alice may consider adding a missing call to `close` and an exception handling block during the example integration and adaptation.

3 DATA COLLECTION FOR API USAGE

As it is difficult to know desirable or alternative API usage a priori, we design an API usage mining approach, called Maple that scales to massive code corpora such as GitHub. We then inspect the results manually and construct a data set of desirable API usage to be used for the Stack Overflow study in Section 4.

In terms of API scope, we target 100 Java APIs that have corresponding Stack Overflow posts and are popular in terms of frequency. From the Stack Overflow dump taken in October 2016, we scan all API methods appearing in Java code snippets by parsing and extracting method calls. We rank the API methods based on frequency and remove trivial APIs such as `System.out.println`. As a result, we select 70 frequently used API methods in Stack Overflow. They are diverse in terms of domain, including Android, Collection, document processing (e.g., String, XML, JSON), graphical user interface (e.g., swing), IO, cryptography, security, Java runtime (e.g. Thread, Process), database (e.g., SQL), networking, date, and time. The rest 30 come from an API misuse benchmark, MUBench [2] after we exclude those without corresponding SO posts or those patterns that cannot be generalized to other projects.

Given an API method of interest, Maple takes three phases to infer API usage. In Phase 1, Maple extracts 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, 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 [7], to check the semantic equivalence of guard conditions, so that we can accurately estimate

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

$$
\begin{aligned}
sequence &:= \epsilon \mid call\ ;\ sequence \\
&\quad \mid construct\ \{\ ;\ sequence\ ;\}\ ;\ sequence \\
call &:= name(t_1, \ldots t_n)@condition \\
construct &:= \texttt{if} \mid \texttt{else} \mid \texttt{loop} \mid \texttt{try} \mid \texttt{catch}(t) \mid \texttt{finally} \\
condition &:= \text{boolean expression} \\
name &:= \text{method name} \\
t &:= \text{argument type} \mid \text{exception type} \mid *
\end{aligned}
$$

Figure 2: Grammar of Structured API Call Sequences

```
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 }
```

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

the frequency of each unique guard correctly. After API usage patterns are mined for each API, we manually inspect all inferred patterns to construct the data set of desirable API usage. This data set is used to report potential API misuse in the Stack Overflow posts in our study discussed in Section 4.

3.1 Structured Call Sequence Extraction and Slicing on GitHub

Given an API method of interest, Maple searches individual code snippets invoking the same method in the GitHub corpora. Maple scans 380,125 Java repositories on GitHub, collected on September 2015. To scale code search to massive corpora, Maple leverages a distributed software mining infrastructure [8] to traverse the abstract syntax trees (ASTs) of Java files. 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, 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 ";". We resolve the argument types of each API call to distinguish method overloading. In certain cases, the argument consists of a complex expression such as `write(e.getFormat())`, where the partial program analysis may not be able to resolve the corresponding type. In that case, we represent unresolved types with $*$, which can be matched with any other types in the following mining phases. Each API call is associated with a guard condition that protects its usage or `true` if it is not guarded by any condition. Catch blocks are also annotated with the corresponding exception types. We normalize a catch block with multiple exception types such as `catch (IOException | SQLException){...}` to multiple catch blocks with a single exception type such as `catch (IOException){...} catch (SQLException){...}`.

Maple builds the control flow graph (CFG) of a GitHub snippet and identifies the enclosing control constructs [1]. 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.

Maple should filter any statements not related to the API method of interest. For example, API calls related to `Properties` in Figure 3 should be removed, since they are irrelevant to `createNewFile`. Maple performs intra-procedural program slicing to retain only data-dependent statements [37]. 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 1 shows the call sequences extracted from Figure 3 with different k. Setting k to 1 leads to best performance empirically (Section 5).

3.2 Frequent Subsequence Mining

Given a set of structured call sequences from Phase 1, Maple finds common subsequences using BIDE [35]. Computing the common subsequence is widely practiced in the literature of API usage mining [27, 28, 34, 40] and 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 guard condition is done in

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

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 2: 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.

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. When matching API signature, Maple matches $*$ with any other types in the same position in an API call. For example, `write(int,*)` can be matched with `write(int,String)` but will not be matched with `write(String,int)`. Maple ranks a list of sequence patterns based on support. Maple filters invalid sequence patterns that do not follow the grammar in Figure 2, as frequent sub-sequence mining can find invalid patterns with unbalanced brackets such as "`foo@true; }; }`".

3.3 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 call from Phase 1 and clusters them based on semantic equivalence. The guard conditions extracted from GitHub 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 removes project-specific predicates (i.e., predicates that do not mention the receiver object or input arguments of the given API call) by substituting them 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 2 illustrates how we derive the guard condition for `String.substring`. It 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`'s arguments or receiver object, 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, prior work on predicate mining checks syntactic similarity after applying rewriting heuristics [20], for example by converting both `arg0<arg1` and `arg1>arg0` to `arg0<arg1`. To overcome the limitation of relying on syntactic similarity, 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 [7] 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 2 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 [20] cannot reason about the equivalence between `-1<arg0` and `0<=arg0`. However, Maple groups these logically equivalent predicates into the same cluster using the integer theorem prover in Z3.

Maple composes the desired API usage by augmenting common subsequences with common guard conditions 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. Similar to the subsequence mining in Phase 2, Maple uses a minimum support threshold θ to filter infrequent guard conditions.

We bootstrap Maple with both the sequence mining threshold σ and the guard condition mining threshold θ set to 0.5, which means frequent subsequences and guard conditions are reported, only if more than half of relevant code snippets on GitHub include them. If Maple learns no patterns with these initial thresholds, we gradually decrease both thresholds by 0.1 till finding patterns. If the mining process does not terminate after 2 hours, we kill the process and increase both thresholds by 0.1 accordingly, since Maple enumerates too many candidate patterns without returning any results. This threshold adjustment method is proven to be effective to achieve a good precision (73%).

3.4 Manual Inspection of Mined API Usage

Maple scans over 380K GitHub projects and finds an average of 55144 relevant code snippets for each API method, ranging from 211 to 450,358 snippets. This result indicates that massive corpora can provide sufficient code snippets to learn patterns from. Maple produces 245 API usage patterns for the 100 APIs in our study scope. This initial set of patterns may include patterns of no interest for several 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, simple APIs such as `Map.put` may not converge to a common pattern.

Therefore, we manually inspect each of the 245 inferred patterns carefully and exclude incorrect ones based on online documentation and pattern frequencies. The overall precision is 73%, resulting in 180 validated, correct patterns that we can use for the empirical study in Section 4. These 180 validated patterns cover 85 of the 100 API methods. The rest 15 API methods do not converge to any API usage patterns that can be confirmed by online documentation, since they are simple to use and do not require additional guard conditions or additional API calls. For example, `System.nanoTime` can be used stand-alone to obtain the current system time. Even though these 15 API methods do not have any patterns, we still include them in the scope of Stack Overflow study, since they represent a category of simple API methods that programmers are less likely to make mistakes.

During the inspection process, if multiple patterns are inferred, each pattern is annotated as either *alternative* or *required*. A code snippet should satisfy one of the alternative pattern and must satisfy all required patterns. For example, Maple learns both `firstKey()@-rcv.size()>0` and `firstKey()@!rcv.isEmpty()` for `SortedMap.firstKey`. Both of them ensure that a sorted map is not empty before retrieving the first key to avoid `NoSuchElementException`. They are considered alternative to each other. As an example of required patterns, programmers must handle potential `IOException`, when reading from a stream (e.g., `FileChannel`) and close it to avoid resource leaks.

Table 3 shows 25 samples of validated API patterns in 9 domains. The entire set of 180 manually validated API usage patterns is attached as a zip file in the EasyChair submission. Alternative patterns are marked with ♣. Column Description describes each pattern. For instance, `TypedArray` is allocated from a static pool to store the layout attributes, whenever a new application view is created in Android. It should be recycled immediately to avoid resource leaks and GC overhead, as mentioned in the JavaDoc.[3] This pattern is supported by 2126 of 3348 related snippets in GitHub and inferred by Maple (ranked #1).

4 API MISUSE STUDY ON STACK OVERFLOW

We use the data set of validated, desirable API usage patterns from Section 3 and study API misuse in Stack Overflow posts.

4.1 Data Collection

We collect all Stack Overflow posts relevant to the 100 Java APIs in our study scope from the data dump taken in October 2016.[4] We extract code examples in the markdown `<code>` from SO posts with the `Java` tag and consider code examples in the answer posts only, since code appearing in the question posts is buggy and rarely used as examples. We gather additional information associated with each post, including view counts, vote scores (i.e., upvotes minus downvotes), and whether a post is accepted as a correct answer.

We parse the embedded code in the Stack Overflow posts using a customized Java parser for partial programs and retain those code examples calling the API method under focus. Our parser wraps an arbitrary snippet with a mocked class and method header as needed and resolves types and API call targets with the API oracle extracted from JDK and Android SDK [26]. 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 and receiver types respectively. In total, we find 217,818 SO posts with code examples for the 100 APIs in our study scope. Each post has 7644 view counts on average.

Maple checks whether the structured call sequence of a Stack Overflow code example is *subsumed* by the desirable API usage in the data set. 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 examples are generalized in the same manner before checking logical implication using Z3. For a SO post with multiple method-level code snippets, Maple inlines invoked methods before extracting the structured call sequence in order to emulate a lightweight inter-procedural analysis.

4.2 Manual Inspection of Stack Overflow

To check whether Stack Overflow posts with potential API misuse reported by Maple indeed suggest undesirable API usage, 200 random samples of SO posts with reported API usage violations are manually checked by the first author. We read the text descriptions and comments of each post and check whether the surrounding text entails the violated pattern narratively. If there are multiple code snippets in a post, we first stitch them and check them all together as one code example. We also account for aliasing and refactoring during code inspection. We examine whether the reported API usage violation could produce any potential behavior anomalies, such as program crashes and resource leaks on a contrived input data or program state, which could have be eliminated by following the desirable pattern. For short posts, this inspection takes about 5 minutes each. For longer posts with a big chunk of code or multiple code fragments, it takes around 15 to 20 minutes.

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

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

Domain	API	Pattern	Support	Description
Collection	ArrayList.get	loop {; get(int)@arg0<rcv.size(); }	31254	check if the index is out of bounds
	Iterator.next	iterator()@true; loop {; next()@rcv.hasNext(); }	218962	check if more elements exist to avoid `NoSuchElementException`
IO	File.createNewFile	if {; createNewFile()@!rcv.exists(); } ♣	5493	check if the file exists before creating it
	File.mkdir	mkdirs()@true	26343	call `mkdirs` instead, which also create non-existent parent directories
	FileChannel.write	try {; write(ByteBuffer)@true; close()@true; }; catch(IOException) {, } ♣	1267	close the `FileChannel` after writing to avoid resource leak
	PrintWriter.write	try {; write(String)@true; close()@true; }; catch(Exception) {; }	2473	close the `PrintWriter` after writing to avoid resource leak
String	StringTokenizer.nextToken	nextToken()@rcv.hasMoreTokens() ♣	36179	check if more tokens exist to avoid `NoSuchElementException`
	Scanner.nextLine	loop {; nextLine()@rcv.hasNextLine(); }	2510	check if more lines exist to avoid `NoSuchElementException`
	String.charAt	charAt(int)@arg0<rcv.length()	27597	check if the index is out of bounds
Regex	Matcher.find	matcher(String)@true; find()@true	5851	call `matcher` to create a `Matcher` instance first
	Matcher.group	if {; group(int)@rcv.find(); }	16447	check if there is a match first to avoid `IllegalStateException`
Database	SQLiteDatabase.query	query(String,String[],String,String[],String,String,String)@true; close()@true	5563	close the cursor returned by `query` to avoid resource leak
	ResultSet.getString	try {; getString(String)@rcv.next(); }; catch(Exception) {; }	18933	check if more results exist to avoid `SQLException`
	Cursor.close	finally {; close()@rcv!=null; }	15732	call `close` in a finally block
Android	Activity.setContentView	onCreate(Bundle)@true; setContentView(View)@true	56321	always call `super.onCreate` first to avoid exceptions
	TypedArray.getString	getString(int)@true; recycle()@true	2126	recycle the `TypedArray` so it can be reused by a later call
	SharedPreferences.Editor.edit	edit()@true; commit()@true	9650	commit the changes to SharedPreferences
	ApplicationInfo.loadIcon	getPackageManager()@true; loadIcon(PackageManager)@true	400	get a `PackageManager` as the argument of `load`
Crypto	Mac.doFinal	try {; getInstance(String)@true; getBytes()@true; doFinal(byte[])@true; }; catch(Exception) {; }	474	get a `Mac` instance first and convert the input to bytes
	MessageDigest.digest	getInstance(String)@true; digest()@true	7048	get a `MessageDigest` instance first
Network	Jsoup.connect	try {; connect(String)@true; get()@true; }; catch(IOException); { } ♣	376	call `get` to fetch the web content
	URL.openConnection	try {; new URL(String)@true; openConnection()@true; }; catch(Exception) {; }	19056	create a `URL` object first and handle potential exceptions
	HttpClient.execute	new HttpGet(String)@true; execute(HttpGet)@true	2536	create a `HttpGet` object as the argument of `execute`
Swing	SwingUtilities.invokeLater	new Runnable()@true; invokeLater(Runnable)@true	20406	create a Runnable object as the argument of invokeLater
	JFrame.setPreferredSize	setPreferredSize(Dimension)@true, pack()@true	394	call `pack` to update the `JFrame` with the preferred size

Table 3: 25 samples of manually validated API usage patterns after GitHub code mining. We attach a zip file containing the entire data set for all 100 APIs and the list of SO posts with potential API usage violations via EasyChair. We will release the data set, when the paper is accepted for publication

True Positive. 136 out of 200 inspected Stack Overflow posts (68%) contain real API misuse. For instance, the following example demonstrates how to retrieve records from `SQLiteDatabase` using `Cursor` but forgets to close the database connection at the end.[5] Programmers should always close the connection to release all its resources. Otherwise, it may quickly run out of memory, when retrieving a large volume of data from the database frequently.

```
1  public ArrayList<UserInfo> get_user_by_id(String id) {
2      ArrayList<UserInfo> listUserInfo = new ArrayList<UserInfo>();
3      SQLiteDatabase db = this.getReadableDatabase();
4      Cursor cursor = db.query(...);
5
6      if (cursor != null) {
7          while (cursor.moveToNext()) {
8            UserInfo userInfo = new UserInfo();
9            userInfo.setAppId(cursor.getString(cursor.getColumnIndex(
                 COLUMN_APP_ID)));
10           // HERE YOU CAN MULTIPLE RECORD AND ADD TO LIST
11           listUserInfo.add(userInfo);
12         }
13     }
14     return listUserInfo;
15 }
```

In many cases, a code example may function well with some hard-coded inputs, even though it does not follow desirable API usage. For example, programmers should check whether the return value of `String.indexOf` is negative to avoid `IndexOutOfBoundsException`. The Stack Overflow post below does not follow this practice, but still works well with a hard-coded constant, `text`.[6] One can argue that the input data is hard-coded for demonstration purposes only, as the role of Stack Overflow post is to provide a starting point rather than teaching complete details of correct API usage. However, if a programmer reuses this code example and replaces the hard-coded `text` with a function call reading from a `html` file, the reused code may crash if the `html` document does not have an expected element. Therefore, it is still beneficial to inform the users about desirable usage and potential pitfalls, especially for a novice programmer who may not be familiar with the given API.

```
1 String text = "<img src=\"mysrc\" width=\"128\" height=\"92\" border=\"0\"
      alt=\"alt\" /><p><strong>";
2 text = text.substring(text.indexOf("src=\""));
3 text = text.substring("src=\"".length());
4 text = text.substring(0, text.indexOf("\""));
5 System.out.println(text);
```

False Positive. Maple mistakenly detects API misuse in 64 posts. The majority reason is that Maple checks for API misuse via a sequence comparison without deep knowledge of its specification, which is not sufficient in 32 posts. For instance, the following SO post calls `substring` (line 5) without explicitly checking whether the start index (`index+1`) is not a negative number and the end index (`strValue.length()`) is not greater than the length of the string.[7] While Maple warns potential API misuse, according to JDK specifications, `indexOf` never returns a negative integer ≤ -2. Thus, the following code is still safe, because `index+1` is guaranteed to be non-negative. Similarly, `strValue.length()` returns the string's length, which cannot be out of bounds. Such cases require having detailed specifications, such as the return value of `indexOf()` is always ≥1.

```
1 public String getDecimalFractions(BigDecimal value) {
2     String strValue = value.toPlainString();
3     int index = strValue.indexOf(".");
4     if(index != -1) {
5         return strValue.substring(index+1, strValue.length());
6     }
7     return "0";
8 }
```

[5] https://stackoverflow.com/questions/31531250
[6] https://stackoverflow.com/questions/12742734
[7] http://stackoverflow.com/questions/7473462

Second, 24 false positives are correct but infrequent alternatives. Maple does not learn these alternative usage patterns, because they do not commonly appear in GitHub. For example, programmers should first call `new SimpleDateFormat` to instantiate `SimpleDateFormat` with a valid date pattern and then call `format`, which is supported by 18,977 related GitHub snippets. An alternative way is to instantiate `SimpleDateFormat` by calling `getInstance`, as shown in the following SO post.[8] This alternative usage is supported by 360 GitHub snippets and therefore not found due to low frequency.

```
... some other code...
public String toString() {
    Calendar c = new GregorianCalendar();
    c.set(Calendar.DAY_OF_WEEK, this.toCalendar());
    SimpleDateFormat sdf=(SimpleDateFormat)SimpleDateFormat.getInstance();
    sdf.applyPattern("EEEEEEEEE");
    return sdf.format(c.getTime());
}
```

In some SO posts, users explicitly state in surrounding natural language text that the given code example must be improved during integration or adaptation. The following example shows how to load a `Class` instance by name and then cast the class.[9] The author of this post comments that "*be aware that this might throw several Exceptions, e.g. if the class defined by the string does not exist or if* `AnotherClass.classMethod()` *doesn't return an instance of the class you want to cast to.*" Maple still flags the post because of a missing exception handling, since the desirable API usage is not reflected in the embedded code. However, it is certainly possible that SO users will carefully read both the code and surrounding text and investigate how to handle edge cases narrated in the text.

```
Class<?> myclass = Class.forName("myClass_t");
myClass_t myVar = (myClass_t)myclass.cast(AnotherClass.classMethod());
```

Sometimes, Stack Overflow users split a single code example into multiple fragments and provide step-by-step explanation, which is considered as a better way of answering questions in Stack Overflow [19]. Maple may report API misuse if two related API calls are split in different code fragments.[10] This can be addressed by stitching these snippets together during analysis.

Overall, Maple detects API misuse with 68% precision. Even for false positives, we believe it would be beneficial for a user to know how many GitHub snippets follow similar API usage patterns.

4.3 Is API Misuse Prevalent on Stack Overflow?

Maple detects potential API misuse in 66,897 (31%) out of 217,818 Stack Overflow posts in our study. Figure 4 shows the prevalence of API misuse from different domains: Database, IO, and network APIs are often misused, since they often require to handle potential runtime exceptions and close underlying streams to release resources properly at the end. Similarly, many cryptography related posts are flagged as unreliable, due to unhandled exceptions. Stack Overflow posts on string and text manipulation often forget to check the validity of inputs or return values. In contrast, UI design, regular expression, and Android examples do not include much API misuse, because the involved API usage does not involve carefully setting guard conditions or exception handling logic.

[8]https://stackoverflow.com/questions/2243850
[9]https://stackoverflow.com/questions/4650708
[10]https://stackoverflow.com/questions/11552754

We manually investigate the consequences of violating desirable API usage and classify them. Among posts with potential API misuse reported by Maple, 76% could potentially lead to program crashes, e.g., unhandled runtime exceptions. 18% could lead to incomplete action, e.g., not completing a transaction after modifying resources in Android, or not calling `setVisible` after modifying the look and feel of a swing GUI widget. 2% could lead to resource leaks in operating systems, e.g., not closing a stream. We fully acknowledge that not all detected violations could lead to bugs when ported to a target application. To accurately assess the runtime impact of SO code examples, one must systematically integrate SO code snippets to real-world target applications and run regression tests.

Many SO code examples aim to answer a particular programming question. Therefore, the authors of these examples may assume the SO users who posted these questions already know about these APIs and may not include complete details of desirable API usage. However, given that each post has 7,644 view counts on average, some SO users may not have similar background knowledge. Especially for novice programmers, it may be useful to show extra tips about desirable API usage evidenced by a large number of GitHub code snippets. We also observe that SO posts with API misuse are more frequently viewed than those posts without API misuse, 8365 vs. 7276 on average. Therefore, there is an opportunity to help developers consider better or alternative API usage mined from massive corpora, when they stumble upon Stack Overflow posts with potential API misuse.

4.4 What are the characteristics of API misuse?

We classify the detected API misuses into three categories based on the required edits to correct the API misuse.

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 17,432 code examples that do not handle exceptions properly. For example, `Integer.parseInt` may throw `NumberFormatException` if the string does not contain a parsable integer. The following example will crash, if a user enters an invalid integer.[11] A good practice is to surround `parseInt` with a try-catch block to handle the potential exception. Unlike *checked exceptions* such as `IOException`, runtime exceptions such as `NumberFormatException` will not be checked at compile time. In such cases, informing users about which runtime exceptions must be handled commonly based on GitHub mining could be very helpful.

```
Scanner scanner = new Scanner(System.in);
System.out.print("Enter Number of Students:\t");
int numStudents = Integer.parseInt(scanner.nextLine());
```

Missing finally. Clean-up APIs such as `close` should be invoked in a finally block in case an exception occurs before invoking those APIs. 83% of Stack Overflow examples that call `Cursor.close` does

[11]https://stackoverflow.com/questions/3137481

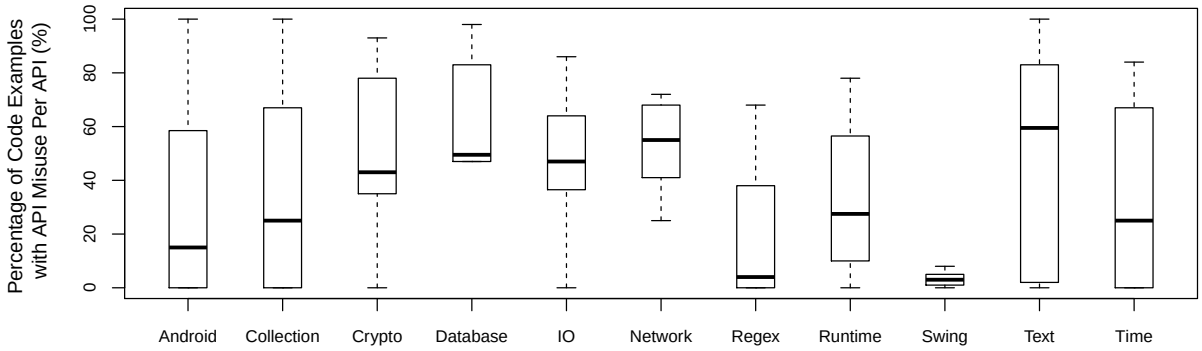


Figure 4: API Misuse Comparison between Different Domains

not call it in a finally block, shown in the following.[12] `Cursor.close` may be skipped, if `getString` (line 5) throws an exception.

```
1 Cursor emails = contentResolver.query(Email.CONTENT_URI,...);
2 while (emails.moveToNext()) {
3     String email = emails.getString(emails.getColumnIndex(Email.DATA));
4     break;
5 }
6 emails.close();
```

Missing if checks. Some APIs may return erroneous values such as null pointers, which must be checked properly to avoid crashing the succeeding execution. For example, `TypedArray.getString` may return null, if the given attribute is not defined in the style resource of an Android application. Therefore, the return value, `customFont` must be checked before passing it as an argument of `setCustomFont` (line 6) to avoid `NullPointerException`, which is violated by the following Stack Overflow example.[13]

```
1 public class TextViewPlus extends TextView {
2   ... some other code ...
3   private void setCustomFont(Context ctx, AttributeSet attrs) {
4       TypedArray a = ctx.obtainStyledAttributes(attrs, R.styleable.
              TextViewPlus);
5       String customFont = a.getString(R.styleable.TextViewPlus_customFont);
6       setCustomFont(ctx, customFont);
7       a.recycle();
8    }
9 }
```

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. The following SO example could throw `BufferUnderflowException`, if the internal cursor already reached the upper bound of the buffer after the `put` operation at line 2.[14] Without resetting the internal cursor, the next `getInt` operation at line 3 would start reading from the upper bound, which is prohibited. We find 7,956 posts that either misses a critical API call or calls APIs in an incorrect order.

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

[12]https://stackoverflow.com/questions/31427468
[13]https://stackoverflow.com/questions/7197867
[14]http://stackoverflow.com/questions/12100651

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 calls `firstKey` on an empty map without a guard, leading to `NoSuchElementException`.[15] Surprisingly, this example is accepted as the correct answer and also upvoted by six other developers on Stack Overflow. We find 12,791 posts with incorrect guard conditions.

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

4.5 Are recognized code examples more reliable?

Within 217,818 studied Stack Overflow posts, 150,294 posts (69%) are accepted as correct answers or have net-positive vote scores (i.e., more upvotes than downvotes). We call such posts as *recognized* posts, since they are endorsed by other SO users and could be considered to have higher quality than others [19]. Overall, 43585 of 150,294 recognized posts (29%) have API misuse (i.e., at least one potential API usage violation), while 22985 of 67524 unrecognized posts (34%) have API misuse. For each API, an average of 37% recognized posts vs. 36% unrecognized posts are detected with potential API usage violations. Figure 5 shows that recognized posts and unrecognized posts do not show much difference in terms of API misuse for the 100 APIs in our study.

Figure 6 shows the percentage of SO posts with different vote scores that are detected with at least one API usage violation. 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.

5 DISCUSSION

Augmentation of Stack Overflow. The study results from Section 4 indicate that even highly voted and frequently viewed SO posts do not necessarily follow desirable API usage. There is an

[15]http://stackoverflow.com/questions/21983867

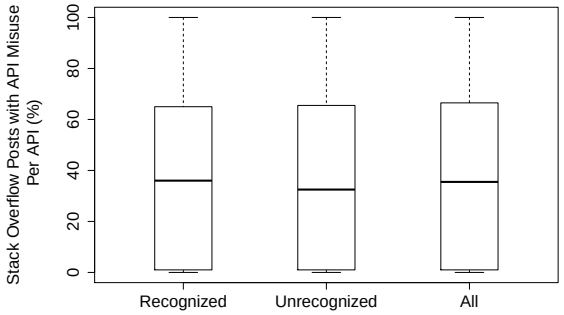


Figure 5: API misuse comparison between *recognized* and *unrecognized* posts on Stack Overflow

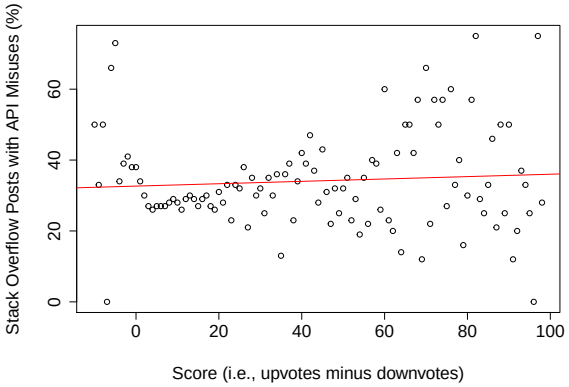


Figure 6: API Misuse Comparison between Code Examples with Different Scores on Stack Overflow

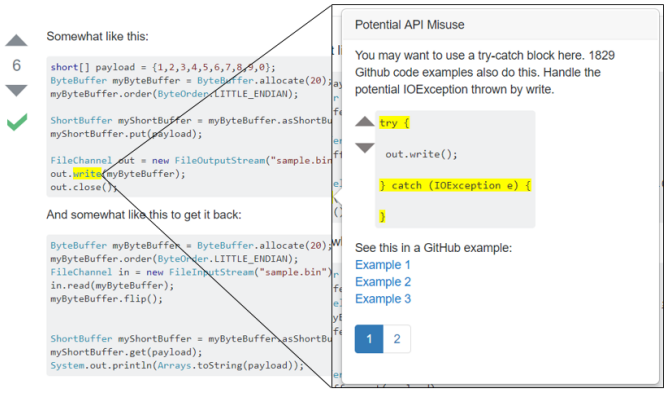
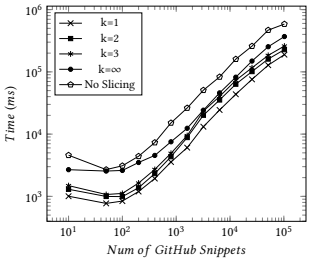


Figure 7: Chrome extension for augmenting Stack Overflow posts with mined API usage

opportunity to help developers consider better or alternative API usage that is mined from massive corpora and is supported by thousands of GitHub snippets. Certainly the goal of Stack Overflow is not to share complete details of how to use a particular



Setting	Precision (%)	Recall (%)	Rank
	Top 5	Top 5	
k = 1	80	91	3
k = 2	79	92	4
k = 3	80	91	4
k = ∞	74	91	4
No Slicing	65	81	9

Figure 8: Mining time and accuracy with varying k bounds.

API or to present compilable, runnable working code. Rather, Stack Overflow often serves the purpose of providing a starting point and helping the user to grasp the gist of how the API works by omitting associated details such as which guard conditions to check and which runtime exceptions to handle. Nevertheless, it would be useful for a user to see related API usage along with concrete examples substantiating the desirable API usage, when the user is browsing the given SO post. Such information may reduce the effort of integrating, adapting, and testing the given code example.

Figure 7 sketches the screen snapshot of a Chrome extension that augments a given SO post. If there exists better or alternative API usage such as enclosing `FileChannel.write` within a try-catch block, the browser extension highlights the relevant API call on the original post and provides a hovering menu with the description *"You may want to use ... 1829 GitHub code snippets also do this"* along with concrete examples. While our proposed browser extension follows a similar style to Codota,[16] Codota does not group related examples based on common API usage, does not quantify how many GitHub code snippets support the common usage, and does not detect API misuse by contrasting the SO post against the usage.

API Usage Mining Running Time and Accuracy. We briefly describe the running time and accuracy of API usage mining employed in Maple. Figure 8(a) shows the performance of Maple with different k bounds. On average, the mining time is within 10 minutes for each API. We run each experiment five times and compute the average execution time. Setting k to ∞ retains all dependent API calls in a sliced call sequence, while setting k to 1 retains only immediately dependent calls. Setting k to 1 can achieve 3.3X speed up compared with setting k to ∞, since it creates shorter call sequences by removing transitively dependent API calls. Maple runs up to 4.6X slower when not removing irrelevant API calls (no slicing).

Figure 8(b) shows the pattern mining accuracy using different k bounds. The evaluation is done for the 30 APIs from MUBench using its ground truth [2]. Maple has 80% precision and 91% recall, when considering top 5 patterns for each API method. 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.

[16] https://www.codota.com/code-browsing-assistant

Threats to Validity. Our study is limited to 100 Java APIs that frequently appear in Stack Overflow and thus may not generalize to other Java APIs or different languages. Our 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 4.2.

6 RELATED WORK

Quality Assessment of Online Code Examples. Prior work has investigated the quality of online code examples from different perspectives. Including code and detailed step-by-step explanations is the key factor in highly voted Stack Overflow posts [19]. 89% of API names in code snippets from online forums are ambiguous and cannot be easily resolved due to the incompleteness of these snippets [6]. SO snippets are often free-standing statements without class or method declarations [25]. About 4% of Java code snippets on Stack Overflow are parsable and only 1% are compilable [39].

Zhou et al. find that 86 of 200 accepted posts on Stack Overflow use deprecated APIs but only 3 of them are reported by other programmers [41]. Fischer et al. investigate security-related code on Stack Overflow and find that 29% is insecure [10]. They further apply clone detection to check whether insecure code is reused from Stack Overflow to Android applications on Google Play and find that insecure code may have been copied to over 1 million Android apps. An et al. investigate copyright issues between Stack Overflow and GitHub [4] and find a large number of potential license violations. Treude and Robillard conduct a survey to investigate comprehension difficulty of code examples in Stack Overflow. The responses from GitHub users indicate that less than half of the SO examples are self-explanatory and the main issues include incomplete code, code quality, missing rationale, code organization, clutter, naming issues, and missing domain information.

While our study also indicates the limitation of code example quality in Stack Overflow, our study focuses on API usage violations that may lead to unexpected behavior such as program crashes and resource leaks by contrasting SO code examples against desirable API usage mined from massive corpora. Our results strongly motivate the need of systematically augmenting Stack Overflow and helping the user to implicitly assess the given SO example by having quantitative evidence about how many GitHub snippets follow (or do not follow) related API usage patterns.

API Usage Mining. There is a large body of literature in mining implicit programming rules, API usage, and temporal properties of API calls. Since API usage mining is only a part of our data set construction process, we are not arguing the novelty of API mining employed in Maple. Nevertheless, we briefly describe how API usage mining employed in Maple is related to prior work.

Gruska et al. extract call sequences from programs and perform formal concept analysis [12] to infer pairwise temporal properties of API calls [15]. Many other specification mining techniques are dedicated to inferring temporal properties of API calls [3, 9, 11, 21, 22, 36]. UP-Miner mines frequent sequence patterns but does not retain control constructs and guard conditions in API usage patterns [34]. Several techniques [17, 18, 29] model programs as item sets and infer pairwise programming rules using frequent itemset mining [14], which does not consider temporal ordering or guard conditions of API calls.

Maple mines from massive corpora of GitHub projects, several orders of magnitude larger than prior work [12]. Maple mines not only API call ordering but also guard conditions by using predicate mining. To our best knowledge, Ramanathan et al. [23] and Nguyen et al. [20] are the only two predicate mining techniques. Ramanathan et al. apply inter-procedure data-flow analysis to collect all predicates related to a call site 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.

7 CONCLUSION

Programmers often resort to code examples on online Q&A forums such as Stack Overflow to learn about how to use APIs correctly during software development. However, the reliability of code examples in Stack Overflow posts is under-investigated. For one hundred Java APIs that are popular and have corresponding SO posts, we mine frequent API usage patterns from 380,125 GitHub repositories, carefully check the resulting 245 mined patterns manually, and construct a data set of 180 validated API usage patterns. Using this data set, we investigate 217,818 Stack Overflow posts and find 66897 posts with potential API usage violations that may produce the symptoms of crash and resource leaks. SO posts in the domain of databases, IO, network, and document manipulation often have API misuse. Even posts with highly voted, accepted answers are not necessarily more reliable than other posts in terms of API usage. This finding demonstrates the prevalence and severity of API misuse in code snippets on Stack Overflow.

Certainly, the purpose of Stack Overflow is to provide a starting point for investigation and its code examples do not necessarily need to include all details of how to reuse the given code. However, for novice developers, it may be useful to show extra tips about desirable API usage evidenced by a large number of GitHub snippets. Our work provides a foundation for a new human-in-the-loop approach to enrich and enhance code snippets included in a collaborative Q&A forum by contrasting them against frequent usage evidenced by many GitHub snippets. Such approach could help the user to implicitly assess the given code example and reduce the effort of integrating, adapting, and testing it in a target application.

REFERENCES

[1] Frances E Allen. 1970. Control flow analysis. In *ACM Sigplan Notices*, Vol. 5. ACM, 1–19.
[2] Sven Amani, Sarah Nadi, Hoan A Nguyen, Tien N Nguyen, and Mira Mezini. 2016. MUBench: a benchmark for API-misuse detectors. In *Proceedings of the 13th International Conference on Mining Software Repositories*. ACM, 464–467.
[3] Glenn Ammons, Rastislav Bodik, and James R. Larus. 2002. Mining specifications. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, New York, NY, USA, 4–16. DOI : http://dx.doi.org/10.1145/503272.503275
[4] Le An, Ons Mlouki, Foutse Khomh, and Giuliano Antoniol. 2017. Stack overflow: a code laundering platform?. In *Software Analysis, Evolution and Reengineering*

(SANER), 2017 IEEE 24th International Conference on. IEEE, 283–293.

[5] Fuxiang Chen and Sunghun Kim. 2015. Crowd debugging. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 320–332.

[6] Barthélémy Dagenais and Martin P Robillard. 2012. Recovering traceability links between an API and its learning resources. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 47–57.

[7] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

[8] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N Nguyen. 2013. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 422–431.

[9] Michael D Ernst, Jake Cockrell, William G Griswold, and David Notkin. 2001. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering* 27, 2 (2001), 99–123.

[10] Felix Fischer, Konstantin Böttinger, Huang Xiao, Christian Stransky, Yasemin Acar, Michael Backes, and Sascha Fahl. 2017. Stack Overflow Considered Harmful? The Impact of Copy&Paste on Android Application Security. In *Security and Privacy (SP), 2017 IEEE Symposium on*. IEEE, 121–136.

[11] Mark Gabel and Zhendong Su. 2010. Online inference and enforcement of temporal properties. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. ACM, 15–24.

[12] Bernhard Ganter and Rudolf Wille. 2012. *Formal concept analysis: mathematical foundations*. Springer Science & Business Media.

[13] Qing Gao, Hansheng Zhang, Jie Wang, Yingfei Xiong, Lu Zhang, and Hong Mei. 2015. Fixing recurring crash bugs via analyzing q&a sites (T). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, 307–318.

[14] Gösta Grahne and Jianfei Zhu. 2003. Efficiently using prefix-trees in mining frequent itemsets.. In *FIMI*, Vol. 90.

[15] Natalie Gruska, Andrzej Wasylkowski, and Andreas Zeller. 2010. Learning from 6,000 projects: lightweight cross-project anomaly detection. In *Proceedings of the 19th international symposium on Software testing and analysis*. ACM, 119–130.

[16] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*, Vol. 1. 2073–2083.

[17] Zhenmin Li and Yuanyuan Zhou. 2005. PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. In *ACM SIGSOFT Software Engineering Notes*, Vol. 30. ACM, 306–315.

[18] Martin Monperrus, Marcel Bruch, and Mira Mezini. 2010. Detecting missing method calls in object-oriented software. In *European Conference on Object-Oriented Programming*. Springer, 2–25.

[19] Seyed Mehdi Nasehi, Jonathan Sillito, Frank Maurer, and Chris Burns. 2012. 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, 25–34.

[20] Hoan Anh Nguyen, Robert Dyer, Tien N Nguyen, and Hridesh Rajan. 2014. Mining preconditions of APIs in large-scale code corpus. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 166–177.

[21] Michael Pradel and Thomas R Gross. 2009. 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, 371–382.

[22] Michael Pradel, Ciera Jaspan, Jonathan Aldrich, and Thomas R Gross. 2012. Statically checking API protocol conformance with mined multi-object specifications. In *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 925–935.

[23] Murali Krishna Ramanathan, Ananth Grama, and Suresh Jagannathan. 2007. Static specification inference using predicate mining. In *ACM SIGPLAN Notices*, Vol. 42. ACM, 123–134.

[24] Caitlin Sadowski, Kathryn T Stolee, and Sebastian Elbaum. 2015. How developers search for code: a case study. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 191–201.

[25] Siddharth Subramanian and Reid Holmes. 2013. Making sense of online code snippets. In *Proceedings of the 10th Working Conference on Mining Software Repositories*. IEEE Press, 85–88.

[26] Siddharth Subramanian, Laura Inozemtseva, and Reid Holmes. 2014. Live API documentation. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 643–652.

[27] Suresh Thummalapenta and Tao Xie. 2007. 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, 204–213.

[28] Suresh Thummalapenta and Tao Xie. 2009. Mining exception-handling rules as sequence association rules. In *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 496–506.

[29] Suresh Thummalapenta and Tao Xie. 2011. Alattin: mining alternative patterns for defect detection. *Automated Software Engineering* 18, 3 (2011), 293.

[30] Emina Torlak and Satish Chandra. 2010. Effective Interprocedural Resource Leak Detection. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1 (ICSE '10)*. ACM, New York, NY, USA, 535–544. DOI : http://dx.doi.org/10.1145/1806799.1806876

[31] Christoph Treude and Martin P Robillard. 2016. Augmenting API documentation with insights from Stack Overflow. In *Proceedings of the 38th International Conference on Software Engineering*. ACM, 392–403.

[32] Medha Umarji, Susan Elliott Sim, and Crista Lopes. 2008. Archetypal internet-scale source code searching. In *IFIP International Conference on Open Source Systems*. Springer, 257–263.

[33] Ganesha Upadhyaya and Hridesh Rajan. 2017. On accelerating ultra-large-scale mining. In *Proceedings of the 39th International Conference on Software Engineering: New Ideas and Emerging Results Track*. IEEE Press, 39–42.

[34] Jue Wang, Yingnong Dang, Hongyu Zhang, Kai Chen, Tao Xie, and Dongmei Zhang. 2013. Mining succinct and high-coverage API usage patterns from source code. In *Proceedings of the 10th Working Conference on Mining Software Repositories*. IEEE Press, 319–328.

[35] Jianyong Wang, Jiawei Han, and Chun Li. 2007. Frequent closed sequence mining without candidate maintenance. *IEEE Transactions on Knowledge and Data Engineering* 19, 8 (2007).

[36] Andrzej Wasylkowski, Andreas Zeller, and Christian Lindig. 2007. Detecting object usage anomalies. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM, 35–44.

[37] Mark Weiser. 1981. Program slicing. In *Proceedings of the 5th international conference on Software engineering*. IEEE Press, 439–449.

[38] Edmund Wong, Jinqiu Yang, and Lin Tan. 2013. Autocomment: Mining question and answer sites for automatic comment generation. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*. IEEE, 562–567.

[39] Di Yang, Aftab Hussain, and Cristina Videira Lopes. 2016. From query to usable code: an analysis of stack overflow code snippets. In *Proceedings of the 13th International Workshop on Mining Software Repositories*. ACM, 391–402.

[40] Hao Zhong, Tao Xie, Lu Zhang, Jian Pei, and Hong Mei. 2009. MAPO: Mining and recommending API usage patterns. In *European Conference on Object-Oriented Programming*. Springer, 318–343.

[41] Jing Zhou and Robert J Walker. 2016. 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, 266–277.