Exposing Library API Misuses via Mutation Analysis

Ming Wen*, Yepang Liu†‖, Rongxin Wu*, Xuan Xie‡, Shing-Chi Cheung* and Zhendong Su§¶
*Department of Computer Science and Engineering
The Hong Kong University of Science and Technology, Hong Kong, China
Email: {mwenaa, wurongxin, scc}@cse.ust.hk
†Shenzhen Key Laboratory of Computational Intelligence
Southern University of Science and Technology, Shenzhen, China. Email: liuyp1@sustc.edu.cn
‡Sun Yat-sen University, China. Email: xiex27@mail2.sysu.edu.cn
§ETH Zurich, Switzerland. Email: zhendong.su@inf.ethz.ch ¶UC Davis, USA

***Abstract*—Misuses of library APIs are pervasive and often lead to software crashes and vulnerability issues. Various static analysis tools have been proposed to detect library API misuses. They often involve mining frequent patterns from a large number of correct API usage examples, which can be hard to obtain in practice. They also suffer from low precision due to an over-simplified assumption that a deviation from frequent usage patterns indicates a misuse.**

We make two observations on the discovery of API misuse patterns. First, API misuses can be represented as mutants of the corresponding correct usages. Second, whether a mutant will introduce a misuse can be validated via executing it against a test suite and analyzing the execution information. Based on these observations, we propose MUTAPI, the first approach to discovering API misuse patterns via mutation analysis. To effectively mimic API misuses based on correct usages, we first design eight effective mutation operators inspired by the common characteristics of API misuses. MUTAPI generates mutants by applying these mutation operators on a set of client projects and collects mutant-killing tests as well as the associated stack traces. Misuse patterns are discovered from the killed mutants that are prioritized according to their likelihood of causing API misuses based on the collected information. We applied MUTAPI on 16 client projects with respect to 73 popular Java APIs. The results show that MUTAPI is able to discover substantial API misuse patterns with a high precision of 0.78. It also achieves a recall of 0.49 on the MUBENCH benchmark, which outperforms the state-of-the-art techniques.

***Index Terms*—Mutation Analysis, Library API Misuses**

I. INTRODUCTION

The use of third-party libraries in Java projects is common. According to a recent study [1], a Java project directly depends on 14 different libraries on average. The `Maven` repository [2] has indexed over 8.77 millions third-party libraries. However, correct usages of Application Programming Interfaces (APIs) provided by many third-party libraries are loosely documented or left unrevised after API updates [3]–[5]. As a result, API misuses are pervasive and account for a major cause of software bugs (e.g., performance issues, software crashes and vulnerability issues [6]–[11]). To detect API misuses, various static analysis tools have been proposed [12]–[20]. These tools commonly mine API *usage patterns* from software codebases. Those *frequent* patterns are deemed as correct API usages, whereas deviations from such patterns are regarded as misuses.

A recent study reported that existing static-analysis based API misuse detectors suffer from low recall and precision in practical settings [6]. The study also made suggestions to address the two limitations. First, to improve recall, existing detectors need to *mine frequent patterns from more correct API usage examples*. However, it is difficult to obtain sufficient correct API usage examples in practice, especially for newly released libraries. Second, to improve precision, the detectors need to go beyond the oversimplified assumption that *a deviation from frequent usage patterns is a misuse*. An uncommon usage of an API is not necessarily an incorrect usage.

In this study, we propose to approach the problem of API misuse patterns' discovery from a new perspective through *mutation analysis* [21]. Mutation analysis has been widely used in software debugging and testing [21]. It mainly contains two steps. First, it creates substantial *mutants* by making small modifications to a program with a set of well-defined *mutation operators* that mimic different kinds of programming mistakes. Second, it runs a given test suite on the mutants and collects execution information for various quality analyses [22]–[26].

Our solution is inspired by two observations. First, various misuses of an API can be viewed as the mutants of this API's correct usages. Therefore, API misuses can be created via applying specifically-designed mutation operators on correct API usages. The design of these mutation operators can be guided by the characteristics of common API misuse patterns, which have been well investigated by existing studies [4], [6]. For instance, one misuse pattern is *missing exception handling* [6]. Many APIs could throw exceptions and the correct usages of these APIs require appropriate exception handling. Given one correct usage of such APIs, we can actively violate the correct usage (e.g., by *deleting the exception handlers*) to create an API misuse. Other patterns of API misuses (e.g., *missing method call* [6]) can be similarly created via applying certain mutation operators (e.g., *deleting method call*). In this way, we can obtain substantial mutants based on an API's correct usages. Unlike existing work [12]–[20], our solution does not require pattern mining from a large number of correct API usages. Instead, it actively creates substantial mutants mimicking API misuses of different patterns. However, deciding if a mutant such created introduces an API misuse is challenging since the manifestation patterns of different API

‖Yepang Liu is the corresponding author

misuses are divergent. Our intuition to address this challenge is that we can validate whether a mutant introduces an API misuse via executing it against the correct usages' test suites and analyzing the execution information. This intuition is inspired by the observation that substantial (42.8%) tests of well-maintained projects would trigger library usages during execution (Section II-C). Therefore, we conjecture that library API misuses can be exposed and validated via running the test suites of client projects (confirmed in Section V-A). In this way, our solution does not make the assumption that a deviation from frequent API usage patterns is a misuse.

In this paper, we propose MUTAPI, an automatic approach that leverages **MUT**ation analysis to discover **API** misuse patterns. To effectively discover API misuse patterns, MUTAPI addresses the following three challenges.

First, discovering API misuse patterns requires MUTAPI to generate mutants that violate existing correct usages, and thus how to effectively generate mutants that mimic API misuses is the key. Conventional mutation operators such as those defined by PIT [27] are less likely to achieve such a goal (Section V-C). To address the challenge, we first investigate how to model correct API usages and then break the modelled usages systematically. Inspired by a recent study [4], MUTAPI models correct API usages as structured call sequences based on a predefined grammar. We then designed eight types of novel mutation operators with the aim to actively violate such modeled usages in a systematic way (Section III-A). With these mutation operators, MUTAPI generates substantial mutants by applying them on certain client projects that use a target API. MUTAPI then runs those mutants against the test suites of the client projects and collects *killing relations*, which includes the *killed mutants* and the corresponding *killing tests*. A mutant is *killed* if its test output differs from that of the original program.

Second, how to validate whether a mutant indeed introduces an API misuse is another challenge. One possible way is to check whether a mutant has been killed by the test suite. However, a mutant can be killed due to multiple reasons (e.g., a logic bug unrelated to any API misuses). How to precisely identify those killing relations arising from API misuses is a key challenging. MUTAPI addresses this challenging via analyzing the failing stack traces of the killing tests (denoted as *killing stack trace*). Specifically, it leverages the killing stack traces to prioritize killing relations based on the following observations. First, given a killing stack trace, the failure's root cause appears closer to the failure point [28]. Therefore, if the top frames of a killing stack trace are library functions, it is more likely to be caused by API misuses (i.e., the root cause resides in the API calls). Second, killing stack traces of a target API should be specific to this API. If a killing stack trace is also observed in the mutation analysis of other APIs, it is less likely to be caused by the target API's misuses. Third, killing stack traces of a target API should not be specific to a certain usage. If a killing stack trace is only observed in the mutation analysis of a specific project, it is more likely to be caused by the bugs specific to this project instead of a general misuse of the target API. MUTAPI leverages these three observations to prioritize observed killing realtions, and assumes those top ranked ones to be caused by the misuses of the target API.

Third, how to effectively distill API misuse patterns after identifying a set of killing relations arising from API misuses is also a challenge. To address this challenge, MUTAPI distills API misuses from substantial killed mutants of the identified killing relations. Specifically, it first models an API misuse as a pair of a *violation type* and an *API usage element*, following the findings of recent studies [4], [6]. It then selects those most frequently observed violation pairs as API misuse patterns.

To evaluate MUTAPI, we selected 73 popular Java APIs collected by recent studies [4], [6], and 16 client projects collected from popular repositories on GitHub. The evaluation results show that MUTAPI is able to achieve a high precision of 0.78 in discovering real misuse patterns of popular APIs. It also achieves a higher recall on the benchmark dataset MUBENCH [6] compared with the state-of-the-art techniques.

In summary, this paper makes the following contributions.

• ***Originality:*** To the best of our knowledge, this is the first study that applies mutation analysis to discover API misuse patterns, and empirical evidences have shown that it is effective in exposing and discovering API misuse patterns.

• ***Implementation:*** We implemented our approach MUTAPI as a tool that can detect API misuse patterns of Java libraries. It employs a set of new mutation operators, which have shown to be effective in dicovering different API misuse patterns.

• ***Evaluation:*** Our evaluation results show that MUTAPI can discover real misuse patterns of popular Java APIs with high precisions. It also detects more misuse instances on the benchmark dataset MUBENCH compared with existing approaches.

II. RELATED WORK AND PRELIMINARIES

In this section, we first introduce related work on mutation analysis and API misuse detection. We then introduce the motivation of this study together with its challenges.

A. Mutation Analysis

Given a program p and a set of mutation operators $\mathcal{O}$, the key idea of mutation analysis is to generate substantial mutants, $\mathcal{M}$, in which each mutant m is a variant of p. The generated mutants are then executed against p's test suite $\mathcal{T}$. A mutant m is *killed* if there exists a test $t \in \mathcal{T}$ whose execution on p and that on m result in different observable final states. Mutation analysis has many applications (e.g., [22], [24]–[26], [29]–[32]), such as evaluating the quality of a test suite (e.g., [22]), test suite reduction (e.g., [30]), improving fault localization (e.g., [23], [26]), security analysis (e.g., [24], [25]), program repair (e.g., [31], [32]) and so on. For instance, to evaluate the quality of a test suite $\mathcal{T}$ [22], *mutation score*, the proportion of killed mutants in $\mathcal{M}$, is computed. The higher the mutation score, the more likely that $\mathcal{T}$ can detect real bugs, and thus the higher quality of the test suite. To the best of our knowledge, *we are the first to apply mutation analysis in discovering API misuse patterns.* In this study, we use k to denote a killing relation that a mutant m is killed by a test t. By executing a test suite $\mathcal{T}$ on all mutants $\mathcal{M}$, we obtain a set of killing relations $\mathcal{K}$.

TABLE I: The Ratio of Test Classes Triggering Target APIs

Project	Apache BCEL	Apache Lang	Apache Text	HttpClient	Lucene-Core
Ratio	0.324	0.483	0.328	0.431	0.573

B. Library API Misuse Detection

Uses of library APIs are often subject to constraints such as call orders and call conditions [6]. Unfortunately, many of such constraints are insufficiently specified by APIs' documentations [3]. As a result, developers also refer to informal references, such as Stack Overflow, to understand the usages of an API [33]. However, as revealed by a recent study [4], code snippets on Stack Overflow can be unreliable, even for those accepted and upvoted answers. Violations of the constraints that should be satisfied may induce software bugs [6]–[11] (e.g., crashes and security issues). Therefore, it motivates a wealth of researches on mining and detecting API misuses automatically [12]–[20]. These existing approaches commonly mine frequent API usage patterns and assume an outlier that deviates from frequent patterns as an API misuse. They differ from each other mainly in how to encode API usages and model frequency. For example, PR-MINER encodes API usages as a set of function calls invoked within the same method, and then leverages frequent itemset mining to identify patterns with a minimum support of 15 usages [12]. JADET builds a directed graph based on method call orders and call receivers [14]. In the graph model, a node represents a method call and an edge represents a control flow relation. GROUPMINER creates a graph-based object usage representation to model API usages for each method. It then leverages subgraph mining techniques to detect frequent usage patterns with a minimum support of 6 [16]. DMMC detects missing calls in a set of method invocations triggered by an object [7]. TIKANGA is built on top of JADET [20]. It extends the properties of call orders to general Computation Tree Logic Formulae on object usages. It then leverages model checking to identify those formulae with a minimum support of 20 in a given codebase.

C. Observation and Motivation

As mentioned earlier, the philosophy of mutation analysis is to *mimic* program bugs via applying mutation operators. In this study, we aim to adopt such a philosophy to detect API misuse patterns, which are a common type of program bugs. The idea is inspired by the following two observations.

First, ***various misuses of an API can be represented as mutants of its correct usages***. Let us illustrate this using the code snippet in Figure 1, which is from project `Apache Commons Lang` [34]. It contains a correct usage of API `java.lang.Float.parseFloat`. Based on this usage, we can generate multiple mutants. Two of them are given as examples in Figure 1, where the second one Mutant#2 repesents an API misuse. It is stated in the API's signature that `NumberFormatException` can be thrown. Mutant#2 is obtained by deleting the `try-catch` statement that encloses the API call. It violates the correct usage of `Float.parseFloat` and follows a well-known API misuse pattern, i.e., *missing exception handling* [4], [6].

Second, ***a program's test suite can be leveraged to validate whether a generated mutant of the program indeed misuses an API***. To study the feasibility of leveraging this observation, we randomly selected five popular projects on GitHub and analyzed whether the execution of the associated test suites can trigger popular API calls. Specifically, we selected the 100 popular APIs collected by an existing study [4]. Table I shows the results. On average, the execution of 42.8% of the test classes (i.e., ranging from 32.4% to 57.3% for different projects) triggers at least one of these APIs. For each test class, we further investigated the number of invocations of these APIs. Figure 2 shows the results, which indicate that substantial library APIs are triggered by test executions. Furthermore, 60.0% of these APIs can throw exceptions, over 85.0% of which are `unchecked exceptions`. The findings suggest that if we violate the correct usages of these APIs via applying mutation operators, there is a high probability that the mutants representing API misuses can be detected by the associated test suite in the form of runtime exceptions. For instance, Mutant#2 can be killed by the test `NumberUtilsTest.testToFloatStringF()` via throwing `NumberFormatException`. The corresponding failing stack trace is shown in Figure 3b.

Based on the above observations, we are motivated in this study to leverage mutation analysis on multiple open-source projects in the wild to discover API misuse patterns.

D. Challenges

Applying mutation analysis to discover API misuse patterns needs to address three challenges. First, how to design mutation operators that can effectively mimic API misuses remains unknown. Second, differentiating mutants that induce API misuses from those that do not is non-trivial. As mentioned before, we can leverage test information to identify whether a mutant introduces an API misuse. However, tests might fail due to multiple reasons. The example of Mutant#1 shown in Figure 1, which is not an API misuse, can also be killed by the test `testToFloatStringF()` with the stack trace shown in Figure 3a. Therefore, we cannot simply conclude that a mutant introduces an API misuse by checking whether it is killed by tests designed for the original program. Conceptually, killing relations can arise from three types of causes. The first

```
public static float toFloat(String str, float defaultValue) {
    if (str == null) {
        return defaultValue;
    }
    try {
        return Float.parseFloat(str);
//      return defaultValue;      Mutant#1 Replace Return Expression
    }   catch (final NumberFormatException nfe) {
        return defaultValue;
    }
//  try {                         Mutant#2 Delete Try {} Catch
        return Float.parseFloat(str);
//  } catch (final NumberFormatException nfe) {
//      return defaultValue;
//  }
}
```

Fig. 1: Two Mutants of API Usage of `Float.parseFloat()`

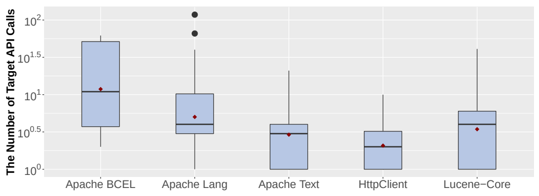
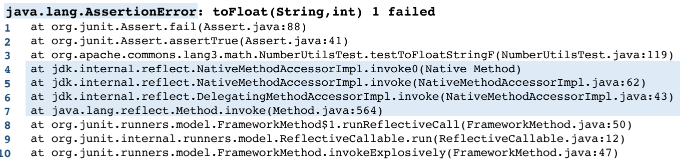
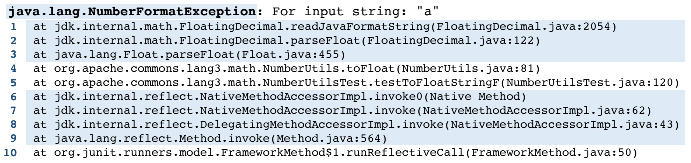


Fig. 2: The Number of Target API Calls Triggered Per Test Class



```
java.lang.AssertionError: toFloat(String,int) 1 failed
  at org.junit.Assert.fail(Assert.java:88)
  at org.junit.Assert.assertTrue(Assert.java:41)
  at org.apache.commons.lang3.math.NumberUtilsTest.testToFloatStringF(NumberUtilsTest.java:119)
  at jdk.internal.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
  at jdk.internal.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
  at jdk.internal.reflect.DelegatingMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:43)
  at java.lang.reflect.Method.invoke(Method.java:564)
  at org.junit.runners.model.FrameworkMethod$1.runReflectiveCall(FrameworkMethod.java:50)
  at org.junit.internal.runners.model.ReflectiveCallable.run(ReflectiveCallable.java:12)
  at org.junit.runners.model.FrameworkMethod.invokeExplosively(FrameworkMethod.java:47)
```

(a) Killing Stack Trace #1 of Mutant #1



```
java.lang.NumberFormatException: For input string: "a"
  at jdk.internal.math.FloatingDecimal.readJavaFormatString(FloatingDecimal.java:2054)
  at jdk.internal.math.FloatingDecimal.parseFloat(FloatingDecimal.java:122)
  at java.lang.Float.parseFloat(Float.java:455)
  at org.apache.commons.lang3.math.NumberUtils.toFloat(NumberUtils.java:81)
  at org.apache.commons.lang3.math.NumberUtilsTest.testToFloatStringF(NumberUtilsTest.java:120)
  at jdk.internal.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
  at jdk.internal.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
  at jdk.internal.reflect.DelegatingMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:43)
  at java.lang.reflect.Method.invoke(Method.java:564)
  at org.junit.runners.model.FrameworkMethod$1.runReflectiveCall(FrameworkMethod.java:50)
```

(b) Killing Stack Trace #2 of Mutant #2

Fig. 3: Two Examples of Killing Stack Traces from Mutants Made on API `Float.parseFloat`

TABLE II: Notations Used in This Study

Notation	Description
$\mathcal{P}$; p	A set of client projects $\mathcal{P}$ and one client project p
$\mathcal{A}$; a	A set of target library APIs $\mathcal{A}$ and API a
$\mathcal{M}$; m	A set of mutants $\mathcal{M}$ and one single mutant m
$\mathcal{T}$; t	A test suite $\mathcal{T}$ and one test case t
$\mathcal{K}$; k	A set of killing relations $\mathcal{K}$ and one killing relation k
$\mathcal{S}$; s	A set of stack traces $\mathcal{S}$ and one stack trace s
a_{target}	The target API for mutation testing
$k_{\langle p,a,m,t \rangle}$	Mutant m made on API a of project p killed by t
s_k	The failing stack traces observed in killing relation k

TABLE III: Grammar of Structured API Call Sequences

sequence	::=	ϵ | *call* ; *sequence* | `if` (*checker*) {} ; *sequence*
	|	*structure* {; *sequence* ; } ; *sequence*
call	::=	$API(v_1, .., v_i, .., v_n)$ | v_{rev}=$API(v_1, .., v_i, .., v_n)$
structure	::=	`if` (*cond*) | `else` | `loop` | `try` | `catch`(v_e)|`finally`
cond	::=	condition expression | *call* | `true` | `false`
checker	::=	*cond* involves receiver v_{rcv} or parameter v_i
API	::=	method name
v	::=	variable | exception | $*$

type resides in the library (Type 1), which indicates that the test fails due to the buggy implementation of the API. The other two types of causes reside in the client project, which indicates that the client program is "buggy" after applying mutation operators. Among them, one type of "bug" is caused by misuses of the API (Type 2) while the other is not (Type 3). Useful API misuse patterns can only be mined from Type 2 "bugs" (e.g., Mutant#2 in Figure 1), and patterns mined from Type 3 "bugs" (e.g., Mutant#1 in Figure 1) will result in false positives. Distinguishing the root cause for a killing relation is challenging. Third, even if we can successfully identify a set of mutants that introduce API misuses, generalizing these mutants to API misuse patterns is non-trivial.

III. MutApi Approach

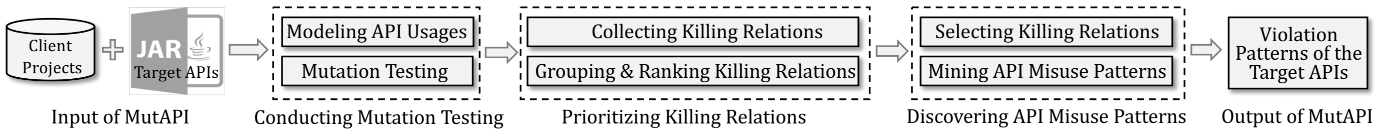
This section presents our approach, the overview of which is shown in Figure 4. The input of MutApi is a set of client projects (i.e., including source code and the associated test suite) and a set of target APIs. The analysis process consists of three main steps. First, MutApi generates mutants by applying a predefined set of mutation operators on the target APIs' usages in the client projects and then runs the tests. Second, it collects the killing relations and prioritizes these relations with respect to each target API. Third, it selects the top ranked killing relations and mines API misuse patterns from the associated killed mutants. The output of MutApi is a list of misuses of the target APIs. The following subsections introduce the details of each step. To ease presentation, Table II summarizes the notations used in this study.

A. Conducting Mutation Testing

A set of mutation operators, which can systematically violate correct API usages, is desired in order to apply mutation analysis to discover API misuse patterns. Conventional mutation operators (e.g., those defined in Pit [27]) are less likely to achieve the goal. For instance, one major type of API misuses is *missing control statements* [4], [6], e.g., missing *exception handling* or *if check* statements. Conventional operators focus on mutating *conditional* or *mathmatics* operators [27], and are not designed to manipulate such control statements. Therefore, they cannot effectively mimic such misuse patterns.

Motivated by a recent study [4], MutApi models *correct API usages* as structured call sequences to effectively mimic various types of API misuses. Such structured call sequences abstract away the syntactic details such as variable names, but keep the temporal ordering, control structures and checkers of API calls in a compact manner [4]. We adapt the grammar defined by the study [4] to represent structured call sequences, which is shown in Table III. A structured call sequence consists of several API calls, each of which can be guarded by structure statements (e.g., `try-catch`) or followed by checker statements (e.g., `null pointer` checker). Overloaded API calls are differentiated via considering the parameters and their types. Figure 5 shows an usage of the class `Iterator`. Suppose our target API (i.e., a_{target}) is `Iterator<>.next()` at line 5. To model the usage of a_{target}, MutApi first identifies the object (e.g., `iterator`), on which a_{target} is invoked, and then identifies other APIs invoked by this object within the same method. In this example, API `Iterator<>.hasNext()` triggered at line 4 will be included in the modeled sequence. MutApi then conducts program slicing [35] (both backward and forward) to extract those structure and checker statements for each of the API call based on its receiver variables v_{rev} (e.g., `value` is the receiver variable at line 5) and parameters v_i. As a result, the `if` statement at line 7 will also be included in the modeled sequence. Note that MutApi only identifies those statements that directly depend on the variables involved in API calls because such a setting is able to achieve the best performance in modeling API usages for misuse detection according to an existing work [4]. As a result, the `if` checker at line 9 will not be sliced into the structured sequence since the checked variable `result` does not directly depend on the receiver variable `value` of our target API. Therefore, the modeled structured sequence for the API usage in Figure 5 is "`if (hasNext()) {;` r_{rev}`=next(); }; if (`r_{rev}`) {};`". The delimiter ";" here is a separator in the grammar, which is different from the semi-colon used in Java.

Based on the modeled structured sequences of correct usages of a_{target}, MutApi then tries to break such usages



Client Projects
JAR
Target APIs
Modeling API Usages
Mutation Testing
Collecting Killing Relations
Grouping & Ranking Killing Relations
Selecting Killing Relations
Mining API Misuse Patterns
Violation Patterns of the Target APIs
Input of MutAPI
Conducting Mutation Testing
Prioritizing Killing Relations
Discovering API Misuse Patterns
Output of MutAPI

Fig. 4: Overview of MUTAPI

```
public double foo() {
    Iterator<Double> iterator = getValues();
    Double value = null;
    if (iterator.hasNext()) {
       value = iterator.next();  // our target API
    }
    if (value == null) return getResult(0);
    Double result = getResult(value);
    if (result < 0) return 0;
    return result;
}
```

Fig. 5: An Usage Example of `Iterator<>.next()`

via applying mutation operators systematically with the aim to mimic various kinds of API misuses. Specifically, we designed eight types of mutation operators, as shown in Table IV, guided by the common characteristics of API misuse patterns [4], [6]. For the case of *incorrect order of API calls* [4], MUTAPI swaps the orders of two call sequences (Type 1), deletes an API call (Type 2) and inserts a new API call (Type 3). For the case of *missing checkers* [6], MUTAPI deletes a checker if there is one (Type 4). For the case of *missing control structures* [4], [6], MUTAPI deletes the structures but keeps the enclosing API calls (Type 5) or deletes the structures together with the enclosing API calls (Type 6). For the case of *missing correct condition* [4], [6], in a checker or an `if` statement, MUTAPI randomly replaces the condition expression with other condition expressions or `boolean` values (Type 7). MUTAPI also mutates the arguments of API calls (Type 8). It replaces the arguments of an API call with other compatible variables (Type 8.1), inserts an argument (Type 8.2) or deletes an argument (Type 8.3) to change the original method call to a call of an overloaded method.

MUTAPI adopts an evolutionary process to generate mutants randomly as described in Algorithm 1. Specifically, it applies at most N mutation operators to the original program to generate a mutant. Certain operators (e.g., replace one condition with another) require necessary code ingredients. In such cases, MUTAPI searches from the original program to select appropriate code elements randomly. By default, N is set to 1, and the effects of N is discussed in Section VI-A.

Algorithm 1: Generating Mutants

```
input : $p$: the original program that has usages of $a_{target}$
input : $\mathcal{O}$: the predefined eight types of mutation operators
input : $Max_{iter}$: the maximum number of iterations (set to $10^5$)
input : $N$: the maximum number of operations to be applied
output: $\mathcal{M}$: a set of generated mutants
1  $\mathcal{M} \leftarrow p$; iter $\leftarrow 0$
2  while iter ++ $< Max_{iter}$ do
3      $m \leftarrow$SelectAnMutantRandomly ($\mathcal{M}$)
4      if GetNumberOfAppliedOperators ($m$) $< N$ then
5          $o \leftarrow$SelectOneMutationOperatorRandomly ($\mathcal{O}$)
6          $i \leftarrow$SelectOneCodeIngredientRandomly ($p$)
7          $m_{new} \leftarrow$GenerateNewMutant ($m, o, i$)
8          if $\mathcal{M}$ does not contain $m_{new}$ then
9              $\mathcal{M} \leftarrow \mathcal{M} \cup m_{new}$
10         end
11     end
12 end
```

B. Prioritizing Killing Relations

MUTAPI collects plenty of killing relations $\mathcal{K}$ after executing all mutants $\mathcal{M}$ against the test suites. Specifically, a killing relation k is collected if mutant m with respect to API a in client project p is killed by p's test case t. In particular, k is also denoted as $k_{\langle p,a,m,t \rangle}$. As described in Section II-D, there are multiple inducing factors for a killing relation, and determining whether the failure of the killing test t is caused by the misuse of the target API a_{target} is challenging. To tackle this problem, MUTAPI leverages the *killing stack traces* s_t to prioritize those killing relations induced by API misuses. For example, Figure 3 shows the top 10 frames of the two killing stack traces for the two mutants shown in Figure 1, respectively. One is induced by API misuse (i.e., missing exception handling), and fails due to `java.lang.NumberFormatException`. The other is induced by general errors unrelated to API misuses, and fails due to `java.lang.AssertionError`. MUTAPI differentiates those killing stack traces that are induced by API misuses from those that are not through prioritization based on stack traces.

The prioritization is based on the analysis of substantial killing stack traces collected from multiple client projects $\mathcal{P}$. To enable such cross-project analysis, MUTAPI preprocesses these killing stack traces to remove frames related to client projects (e.g., Frame 4 in Figure 3b) or the `JUnit` framework (e.g., Frame 1 in Figure 3a) since these frames are unlikely to characterize the patterns of stack traces induced by misuses of library APIs. As a result, only those frames displayed in blue background in Figure 3 are kept for further analysis. The failing signature (e.g., `java.lang.AssertionError` or `java.lang.NumberFormatException`) is also important to understand the failure cause. Therefore, we also keep such information in the processed stack traces. However, project-specific information (e.g., `toFloat()` in stack trace #1 or *for input string: "a"* in stack trace #2) has been filtered out. Two killing stack traces are regarded as the same if both their failing signature and the processed frames are the same.

After preprocessing each killing stack trace, MUTAPI obtains a set of unique traces $\mathcal{S}$. MUTAPI then groups those killing relations whose killing stack traces are the same. With respect to a_{target}, MUTAPI then tries to identify those killing stack traces in $\mathcal{S}$ that are induced by the misuse of a_{target} (denoted as *target API misuse induced killing stack traces*) via prioritization based on the following insights.

Target API misuse induced killing stack traces should not be specific to a certain usage. If a killing stack trace is indeed induced by the misuse of a_{target}, it should be observed in the mutation analysis of multiple usages of a_{target} across different projects. Otherwise, such a stack trace is more likely to be caused by the bugs specific to an API usage in a specific

TABLE IV: Mutation Operators to Violate API Usages

Type	Designed Mutation Operator			Description
1	$sequence_1$; $sequence_2$	$\Rightarrow$	$sequence_2$; $sequence_1$	*Swapping API call sequences*
2	$sequence_1$	$\Rightarrow$	$sequence_1$; $sequence_2$	*Adding an API call to a call sequence*
3	$sequence_1$; $sequence_2$	$\Rightarrow$	$sequence_2$	*Deleting an API call from a call sequence*
4	`if` $(checker)\ \{\}$; $sequence$	$\Rightarrow$	$sequence$	*Deleting the checker of receivers or parameters*
5	$structure\ \{\ sequence_1\ ;\ \}$; $sequence_2$	$\Rightarrow$	$sequence_1$; $sequence_2$	*Deleting the control structure of a sequence*
6	$structure\ \{\ sequence_1\ ;\ \}$; $sequence_2$	$\Rightarrow$	$sequence_2$	*Deleting the control structure together with the enclosing APIs of a sequence*
7	`if` $(cond_1 \vert checker)$	$\Rightarrow$	`if` $(cond_2 \vert$ `true` $\vert$ `false`$)$	*Changing the control condition to other conditions or boolean values*
8.1	$API(t_1, ..., t_i, ..., t_n)$	$\Rightarrow$	$API(t_1, ..., t_j, ..., t_n)$	*Replacing arguments (from t_i to t_j) of a method call*
8.2	$API(t_1, ..., t_i, ..., t_n)$	$\Rightarrow$	$API(t_1, ..., t_i, ..., t_{n+1})$	*Inserting arguments of a method call (changing API to an overloaded method)*
8.3	$API(t_1, ..., t_i, ..., t_n)$	$\Rightarrow$	$API(t_1, ..., t_i, ..., t_{n-1})$	*Deleting arguments of a method call (changing API to an overloaded method)*

project instead of general misuses of a_{target}. Therefore, MUTAPI measures the ratio of usages whose mutation analyses have observed the killing stack trace s to prioritize whether s is caused by the misuse of a_{target} as follows:

$$freq(s) = \frac{|\{u_i | u_i \in \mathcal{U}, mutation\ analysis\ of\ u_i\ observed\ s\}|}{|\{u_i | u_i \in \mathcal{U}, u_i \text{ is the } i_{th} \text{ usage of } a_{target}\}|} \quad (1)$$

where $\mathcal{U}$ contains all the usages of a_{target} extracted from all the client projects $\mathcal{P}$.

Target API misuse induced killing stack traces should be specific to this API. If a killing stack trace is induced by the misuse of a target API, it should not be observed in the mutation analysis of other APIs. Otherwise, such a stack trace is less likely to be induced by the misuses of the target API since it is general to multiple divergent APIs. Motivated by this, MUTAPI measures the inverse frequency of s, which is the ratio of the number of s observed in the mutation analysis of a_{target} to that of all target APIs $\mathcal{A}$, as follows:

$$inverseFreq(s) = \frac{|\{k_{\langle p, a_{target}, m, t\rangle} | k \in \mathcal{K}, s_k = s\}|}{|\{k_{\langle p, a_i, m, t\rangle} | k \in \mathcal{K}, a_i \in \mathcal{A}, s_k = s\}|} \quad (2)$$

where $k_{\langle p, a_{target}, m, t\rangle}$ is a killing relation observed in the mutation analysis of the target API a_{target}, while $k_{\langle p, a_i, m, t\rangle}$ is a killing relation observed in that of another API a_i.

The top frames of the target API misuse induced killing stack traces should be the target library's functions. As revealed by existing studies, in a stack trace, functions where the root causes reside appear closer to the failure point [28]. The top frames of stack trace #2 shown in Figure 3a are functions from the library containing the target API `java.lang.Float.parseFloat`. However, the top frames of stack trace #1 are those functions related to the `JUnit` framework. Here, we examine the ***original stack traces*** instead of the processed ones in order to investigate the position of each frame at the failure time. Therefore, killing stack trace #2 is more likely to be induced by the misuse of API `java.lang.Float.parseFloat`. Motivated by these observations, for a given stack trace, we propose to use the rankings of those frames from the target library to approximate its likelihood to be induced by misusing the library APIs.

$$likelihood(s) = \sum_{i=1}^{n} library(f_i) * 1/i \quad (3)$$

where f_i represents the i_{th} frame of stack trace s. Function $library(f_i)$ returns 1 if f_i comes from the target library and 0 otherwise. For the stack trace shown in Figure 3a, the likelihood 0.76, which is computed as $\frac{1}{4} + \frac{1}{5} + \frac{1}{6} + \frac{1}{7}$.

MUTAPI prioritizes all the unique stack traces $\mathcal{S}$ with respect to the target API a_{target} via computing the following score for each stack trace s.

$$score(s) = freq(s) * inverseFreq(s) * likelihood(s) \quad (4)$$

C. Discovering API Misuse Patterns

For each unique killing stack trace s in $\mathcal{S}$, MUTAPI computes its score with respect to a_{target}. MUTAPI then retrieves all killing relations whose killing stack traces are the same as s. Let $\mathcal{K}_s$ denote such a set of killing relations, where $\mathcal{K}_s$=$\{k | k \in \mathcal{K}, s_k = s\}$. MUTAPI distills API misuse patterns from those killed mutants $\mathcal{M}_s$ associated with $\mathcal{K}_s$. To effectively distill API misuses, MUTAPI models an API misuse as a violation pair, p = ⟨*violation type*, *API usage element*⟩, following a recent work [6]. Specifically, there are three violation types, *missing*, *redundant* and *incorrect*, according to existing studies [4], [6]. An API usage element can refer to a *method call*, *null checker*, *condition*, *exception handling*, *iteration*, *parameter* and so on [4], [6]. Specifically, MUTAPI leverages the following rules to distill violation pairs from mutants via investigating the applied mutation operators. MUTAPI distills violation pairs of type *missing* from mutants created by mutation operators #3, #4, #5, #6, and #8.3 since they delete code elements; violation pairs of type *redundant* are distilled from mutants created by mutation operators #2 and #8.2 since they add code elements; and violation pairs of type *incorrect* are distilled from mutation operators #1, #7 and #8.1 since they replace existing code elements with others. For *API usage element*, MUTAPI distills it via analyzing the code element that has been mutated. For instance, the second mutant shown in Figure 1 is generated by applying mutation operator #5. Therefore, the violation type is *missing*. The code element being mutated is a `try-catch` statement. As a result, the distilled violation pair is ⟨*missing*, *exception handling*⟩.

Multiple violation pairs can be distilled based on those mutants $\mathcal{M}_s$. Therefore, MUTAPI further prioritizes these pairs based on their frequencies. Specifically, MUTAPI identifies a set of unique pairs $\mathcal{P}_{\mathcal{M}_s}$ based on $\mathcal{M}_s$. It records the number of occurrences for each vp in $\mathcal{P}_{\mathcal{M}_s}$ (i.e., denoted as $count(vp)$). MUTAPI then prioritizes all the violation pairs based on their occurrences among all the pairs distilled:

$$ratio(vp) = \frac{count(vp)}{\sum_{vp_i \in \mathcal{P}_{\mathcal{M}_s}} count(vp_i)} \quad (5)$$

Finally, the suspicious score for a violation pair vp to be the misuse of the target API a_{target} is computed as:

$$suspicious(vp, a_{target}) = score(s) * ratio(vp) \quad (6)$$

where $score(s)$ measures the likelihood of those killing relations $\mathcal{K}_s$ sharing the same stack trace s to be induced by the misuses of a_{target}, and $ratio(vp)$ measures the frequency of the violation pair vp observed among all violation pairs distilled from those killed mutants associated with $\mathcal{K}_s$.

IV. Experiment Setup

This section presents our experiment setup and the research questions to be investigated.

A. Target APIs Selection

To evaluate the effectiveness of MutApi, we selected 73 popular Java APIs collected by a recent study [4] for experiments[1]. Among them, 43 are the most frequently discussed APIs in Stack Overflow (e.g., `Iterator<>.next()`). These APIs are often used in practice, and developers are frequently confused by their usages. The remaining 30 APIs come from MuBench [6], which is a benchmark dataset of API misuses. These APIs are in diverse domains, and the majority of them come from four categories: Common Library (e.g., math, collection, time, xml), GUI (e.g., Swing), Security (e.g., `java.security.Cipher()`) and Database.

B. Client Projects Selection

In order to discover API misuse patterns, MutApi requires a set of client projects to perform mutation analysis. In this study, we selected 16 open-source Java projects from four different categories as shown in Table V. These projects are selected randomly from GitHub that satisfy the following two criteria. First, the number of unique target APIs that it covers should be greater than 15. Since MutApi aims at detecting misuses with respect to the selected target APIs, the selected client projects should contain as many usages of those APIs as possible. It is hard to find a client project that triggers all the 73 target APIs since those APIs are from diverse domains. Therefore, we set a reasonable threshold of 15. Our second criterion is that the mutation coverage (measured using Pit [27]) of a client project should be greater than 0.70. We set this threshold to ensure the test suite's quality [39] since MutApi relies on the associated test suite to validate whether a mutant introduces an API misuse. We will further discuss the effect of the test suite's quality on the performance of MutApi in Section VI. With the two criteria, we randomly selected four Java projects from each of the four categories of domains where the 73 APIs commonly come from (i.e., GUI, Library, Security and Database). The categorization of the client projects is based on the Apache official definition [40] and GitHub Topics [41]. In total, the selected 16 projects cover 55 distinct APIs among the selected 73 APIs. The remaining 18 APIs are not covered because they are rarely used by large-scale and popular projects (e.g., `jsoup.Jsoup.connect()`).

[1]The prior study collected 100 popular APIs [4]. We excluded 27 Android APIs in our study because the mutation analysis for Android apps is different from that for general Java projects [36]–[38] (e.g., the former requires mutation operators to mutate the `AndroidManifest.xml` file)

TABLE V: Selected Client Projects

Category	Project	#Covered API	#Source	#Test	#KLOC
GUI	Apache **FOP** [42]	35	1577	391	363.2
	SwingX [43]	28	551	340	215.4
	JFree**Chart** [44]	19	637	350	294.8
	iTextPdf [45]	34	894	609	298.6
Library	Apache **Lang** [34]	24	153	174	144.1
	Apache **Math** [46]	19	826	498	307.1
	Apache **Text** [47]	17	85	63	40.8
	Apache **BCEL** [48]	23	417	75	75.7
Security	Apache **Fortress** [49]	22	214	74	122.2
	Santuario [50]	24	467	198	135.6
	Apache **Pdfbox** [51]	28	598	103	154.1
	Wildfly-Eytron [52]	36	763	151	161.3
Database	Jack**Rabbit** [53]	34	2443	646	611.4
	Apache **BigTop**	22	176	52	20.4
	H2Database [54]	20	566	317	305.4
	Curator [55]	18	197	52	20.4

C. Research Questions

We study three research questions to evaluate MutApi:

- **RQ1:** Can MutApi discover API misuse patterns? What kinds of misuse patterns can be more easily detected by it?

MutApi generates a ranked list of API misuse patterns. In RQ1, we investigate whether those top-ranked API misuse patterns detected by MutApi are *true positives*. Specifically, we use the metric *Precision@N* to evaluate the performance of MutApi. *Precision@N* reports the percentage of true positives among the top N API misuse patterns ($N = 1, 5, 10, ...$) reported by MutApi. To judge whether a detected misuse pattern is a true positive or not, we follow the strategy adopted by an existing study [4]. Specifically, we manually inspected it based on online documentations (i.e., whether the misuse has been documented online) and existing literature (e.g., whether the misuse pattern has been reported by existing studies). We then investigate the characteristics of the APIs whose misuse patterns have been detected by MutApi by checking the associated Java documentations.

- **RQ2:** Can MutApi detect API misuse instances on the state-of-the-art benchmark dataset MuBench?

Based on the discovered misuse patterns (i.e., violation pairs), MutApi is able to detect misuse instances. Specifically, MutApi applies the same analysis as described in Section III-A to model usage of an API in the form of Structured API Call Sequences. Then, it checks whether the modelled sequences violate confirmed misuse patterns. We applied MutApi to MuBench [6] to see if it can detect the API misuses in the benchmark dataset. Specifically, we investigate the *recall* of MutApi, i.e., the ratio of the API misuses that can be detected by MutApi among the 53 real instances used in the Experiment R in the MuBench work [6]. We chose those misuses in Experiment R since they are all real misuse instances identified from open-source projects. To compare with existing approaches, we chose four baselines (i.e., Jadet [14], GroupMiner [16], DMMC [7] and TIKANGA [20]) that have been systematically evaluated by existing study [6].

- **RQ3:** Compared with conventional mutation operators, how effective and efficient do our proposed mutation operators perform in detecting API misuse patterns?

In this work, we propose eight types of mutation operators specific for API misuse detection, which are adopted by MutApi in mutation analysis. In this research question, we compare our proposed mutation operators with the traditional

TABLE VI: Generated Mutatns and Killing Relations

Project	#Mutant	#Killing Relation	#Unique Trace	#PitMutant
Fop	4024	6539	64	12845
SwingX	1063	9373	36	12348
Chart	1951	1603	139	6250
iTextPdf	9323	36083	102	11753
Lang	6144	16890	174	6502
Math	1599	34882	42	14726
Text	837	1408	64	6740
Bcel	4072	10338	86	9827
Santuario	2171	22521	27	3909
Pdfbox	4807	45735	29	11381
Fortress	1947	9015	55	3115
Wildfly	2478	6473	40	4091
Rabbit	1431	5033	48	4175
BigTop	937	2107	37	2508
H2	6155	2036	67	9087
Curator	3321	5406	43	5431
Average	3266	13465	65	7793

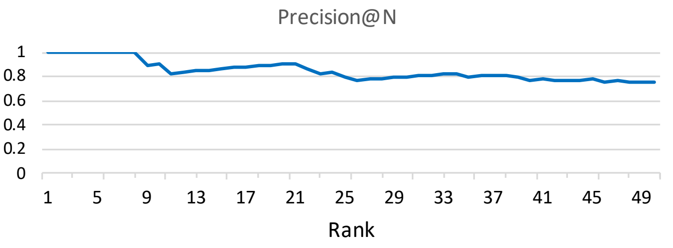


Fig. 6: Precision@N of MUTAPI for the Top-Ranked Misuses

ones used in PIT [27] using two metrics: (1) *efficiency* (i.e., the time required for mutation analysis) and (2) *effectiveness* (i.e., the number of API misuse patterns detected).

V. EXPERIMENTAL RESULTS

A. *RQ1:Effectiveness of* MUTAPI

We applied MUTAPI to all client projects to discover misuse patterns of our selected target APIs. MUTAPI generated lots of mutants and collected substantial killing relations with the associated killing stack traces. Table VI shows the statistics. Specifically, MUTAPI generated 3,266 mutants and collected 13,465 killing relations per client project. After processing the killing stack traces (i.e., removing client project frames), it obtained 65 unique stack traces per client project. MUTAPI then analyzed those killing relations across different client projects and distilled around 300 API misuse patterns (i.e., violation pairs). We manually examined the top 50 patterns following the procedures as described in Section IV-C. Figure 6 shows the results of *Precision@N*. It shows that MUTAPI achieves a high precision of 0.90 within the top 10 candidates. The precision slightly drops to 0.78 within the top 50 ones. Table VII shows several selected top-ranked violation pairs that are real API misuse patterns, which cover different types: *missing checker*, *missing call*, *missing exception*, *incorrect condition* and *redundant call*. These results show that MUTAPI can discover real API misuse patterns with a high precision.

In the detected patterns, missing necessary checker `if (rev==null)` on the receiver of API `Line.intersection()` is ranked at the first place. For the two correct usages of this API in the codebase, one of which is shown in Figure 7, MUTAPI generated mutants (i.e., deleting the checker) that were killed by the associated test suite with `NullPointerException`. The killing stack trace is unique to this API (i.e., the *inverse frequency* is high) whose top frames are method invocations from the target library (i.e., the *likelihood* is high). Furthermore, the killing stack trace has been observed in the mutation analysis for both of the two correct usages (i.e., the *frequency* is high). As a result, missing the checker is deemed as a misuse of `Line.intersection()` with high confidence. This misuse has been confirmed by existing literature [6]. Figure 8 shows another example, which shows a correct usage of the API `Iterator<>.next()`, and the associated killing stack trace after deleting the structure containing API call `Iterator<>.hasNext()`. Such a killing stack trace was observed during the mutation analysis of multiple correct usages across different client projects (e.g., *jfreechart*, *commons-bcel* and *commons-math*).

We further investigated the documentations of APIs whose misuse patterns were discovered by MUTAPI to see whether they share similar characteristics. We found that 78.9% of them will throw *unchecked exceptions*. Figure 9 shows the distributions of these unchecked exceptions. Note that, the sum of the distributions for all the exceptions is greater than 1, since an API might throw multiple types of exceptions. The above results indicate that MUTAPI is more effective in detecting misuses of those APIs that will throw unchecked exceptions. This is because those mutants representing such misuses are more likely to be killed in the form of runtime exceptions (i.e., their killing does not require strong test oracles). There are also other types of APIs whose misuses are less likely to be detected by MUTAPI (see Section VI-B).

B. *RQ2:Performance of* MUTAPI *on* MUBENCH *Benchmark*

Figure 10 shows the recall of MUTAPI and the baselines on the benchmark of MUBENCH. The results of baselines are directly extracted from the previous study [6]. Note that there are two experimental settings for the baseline approaches. One is a practical setting which does not include any hand crafted API usages (denoted as Recall#Practical in Figure 10). The other setting involves hand crafted API correct usages (denoted as Recall#Crafted in Figure 10). The two settings are detailedly

```
// line is an object of type Line passed from the parameters
Cartesian3D v1D = line.intersection(subLine.line);
if (v1D == null) {                                 // Necessary Checker
   return null;
}
Location loc1 = remainingRegion.checkpoint(line.toSubSpace(v1D));
```

Fig. 7: A Correct Usage of API `Line.intersection()`

```
if (!iterator.hasNext()) {
   return EMPTY;
}
final Object first = iterator.next();
```

(a) A Correct Usage

```
-1:java.util.NoSuchElementException
0:java.util.ArrayList$Itr.next()
6:java.lang.reflect.Method.invoke()
```

(b) Killing Stack Trace

Fig. 8: A Correct Usage and the Associated Killing Stack Trace of `Iterator<>.next()`

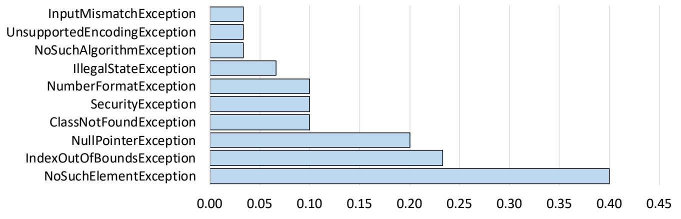


Fig. 9: Distributions of Detected Unchecked Exceptions

TABLE VII: Examples of Top-Ranked Discovered Violation Pairs

Rank	API	VIolation Pair	API Element	Description & Confirming References
1	`rcv = Line.intersection()`[1]	MISSING CHECKER	`if (rcv == null) {}`	The returning value could be null [6]
2	`Iterator<>.next()`	MISSING CALL	`Iterator<>.hasNext()`	Should check if there are sufficient tokens [4], [6]
3	`StringTokenizer.nextToken()`	MISSING CALL	`StringTokenizer.hasMoreTokens()`	Should check if there are sufficient tokens [4], [6]
4	`Integer.parseInt()`	MISSING EXCEPTION	`try {} catch(NumberFormatException)`	Might throw exceptions [6]
5	`Double.parseDouble()`	MISSING EXCEPTION	`try {} catch(NumberFormatException)`	Might throw exceptions [6]
6	`PdfArray.getPdfObject()`[2]	INCORRECT CONDITION	`if (!PdfArray.isEmpty()) {}`	Should check if the object is empty [6]
7	`rcv = SortedMap.firstKey()`	MISSING CHECKER	`if (rcv == null) {}`	The returning value could be null [4], [6]
8	`rcv = StrBuilder.getNullText()`[3]	MISSING CHECKER	`if (rcv == null) {}`	The returning value could be null [6]
10	`MessageDigest.getInstance()`	MISSING EXCEPTION	`try {} catch(NoSuchAlgorithmException)`	Might throw exceptions [56]
12	`Matcher.group()`	MISSING CALL	`Matcher.find()`	Required to be used together [57]
25	`Iterator.next()`	REDUNDANT CALL	`Iterator.remove()`	Shouldn't call `remove` during iteration [6]

1: from library `org.apache.commons.math`; 2: from library `com.itextpdf.text`; 3: from library `org.apache.commons.lang`; the others from Java

explained in the existing study [6] and we do not make further explanations in this study. MUTAPI is not evaluated on the crafted setting since the crafted API usage examples are not equipped with a test suite required for mutation analysis.

As shown in Figure 10, MUTAPI is able to detected 26 out of the 53 real API misuses. It achieves the highest recall of 0.49. In the practical setting, MUTAPI significantly outperforms all the baseline approaches. The second best approach is DMMC, which achieves a recall of 0.21. GROUPMINER is unable to detect any API misuses in the benchmark. One of the major reasons that limit existing approaches in detecting more API misuses is that the number of usage examples in the codebase is too small (below the minimal support values required by existing approaches for pattern mining). For instance, GROUPMINER [16] detects frequent usage patterns with a minimum support of 6 and TIKANGA [14] sets the minimum support to 20. This means that they require at least 6 or 20 usage examples in the codebase. In the crafted setting, since more correct usage examples have been added to the codebase manually, the recall of existing approaches have improved accordingly. Nonetheless, the results of MUTAPI obtained in the practical setting still outperform all the baseline approaches under this crafted setting.

We further investigated the reasons why certain patterns of API misuse instances cannot be detected by MUTAPI and found three major types of reasons: *no correct API usages*, *no mutant coverage* and *inadequate test suite* (see Section VI-B). We plan to evaluate MUTAPI on more datasets other than MUBENCH [6] in the future since one threat to validity of our results here is that the design of our mutation operators was partially inspired by the study of this benchmark.

C. RQ3:Efficiency and Effectiveness of Mutation Operators

Table VI shows the number of mutants generated by MUTAPI and PIT in column *#Mutants* and *#PitMutants*, respectively. Note that when generating mutants using PIT, we only target at those source files that involve usages of our selected target APIs. As shown in the table, on average, PIT generates

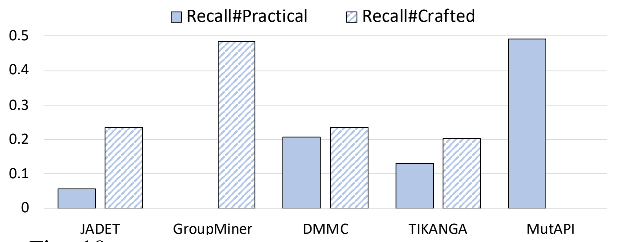


Fig. 10: Recall of MUTAPI and the Baseline Approaches

2.39 times more mutants than MUTAPI. Mutation analysis is known to be computationally expensive since it needs to compile the mutants and execute them against the associalted test suite [58], [59]. As a result, it takes 9.87 minutes for MUTAPI to finish the analysis per project on average while it takes 20.93 minutes for conventional mutation operators.

We fed the mutants generated by PIT and the associated killing relations to MUTAPI. We manually checked the Top-50 ranked results following the same procedure adopted in RQ1. Figure 11 shows the number of real API misuse patterns (i.e., true positives) detected by PIT. In total, PIT is only able to detect 5 true positives of the type *incorrect condition* and *missing call*. This is because that the conventional mutation operators focus on mutating arithmetic and conditional operators [27], which mostly result in assertion errors general to all client projects. Therefore, it is hard to leverage them to distill real API misuses. The conventional mutation operators also mutate conditions (e.g., forcing conditions to `ture` or `false`) and remove void method calls. This explains why PIT discovers five real API misuses.

These results demonstrate that our proposed mutation operators are more effective and efficient in discovering API misuse patterns than conventional ones, which reflect the need to propose domain-specific mutation operators for the discovert of API misuse patterns. Figure 11 also shows the distributions of different types of misuse patterns that are detected by MUTAPI. It indicates that *missing checker*, *missing exception* and *missing call* are the most frequently ones. This is in line with the findings of existing studies [4], [6].

VI. DISCUSSIONS

A. Effects of the Number of Applied Mutation Operators

MUTAPI can apply N mutation operators when generating mutants (see Algorithm 1). By default, N is set to 1. The effect of the maximum number of mutation operators can be applied on the effectiveness of MUTAPI is unknown. We investigate such effects in this subsection.

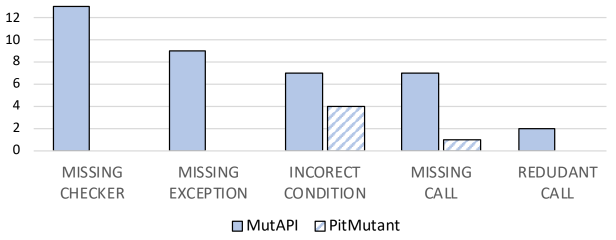


Fig. 11: The Number of True API Misuse Patterns Detected by MUTAPI and PIT among the Top 50 Results

Figure 12 shows the number of mutants generated when N is set to 1 to 5. We can see that, the number of generated mutants significantly grows with the increase of N. Thus, it becomes more expensive to conduct mutation analysis. However, we observe that the number of the unique failing stack traces increases marginally as N grows. For instance, the total number of unique stack traces is 543 when N=1. The number only increases by 5 when N=2, while the number of mutants increases by $20,501$ in total for the 16 client projects. This indicates that applying a single mutation operator is sufficient to discover most of the error spaces (i.e., in the form of failing stack traces) for exposing library misuses.

B. *Limitations of* MUTAPI

There are three major reasons that hinder MUTAPI to discover misuse patterns for a target API.

No correct usages. If there are no correct usages of the target API in the input client projects, MUTAPI cannot discover the corresponding misuses via mutation analysis. For example, in our experiments, we found that there are no correct usages for API `org.kohsuke.args4j.api.Parameters.getParameter()` in all the 16 client projects. We did not pick additional projects that have correct usages of this API in order to guarantee the generality of our results and findings. Thus, MUTAPI did not discover any misuse patterns for this API. Existing approaches also suffer from this limitation as discussed before, and, even worse, they require a larger number of correct usages than our approach.

No mutant coverage. We observe that certain API misuse patterns require specific values that can not be discovered by our approach. For instance, `javax.crypto.Cipher("DES")` is a misuse since it should not be used with the DES mode [60]. However, MUTAPI only searches from the same source file in order to find appropriate code elements when replacing parameters or conditions. As a result, if there is no string "DES" in the same source file, MUTAPI cannot create a mutant which represents this API misuse. Similar cases are observed for other APIs such as `String.getBytes()`.

Inadequate test suite. The effectiveness of MUTAPI is subject to the quality of the associated test suite. Even if MUTAPI has generated a mutant that represents a real API misuse, it still has no chance to detect the misuse if the associated test suite cannot kill the mutant. For example, for the misuse of API `StatisticalCategoryDataset.getDtDevValue()` in MUBENCH, MUTAPI successfully created a mutant that mimicked a misuse. Unfortunately, it could not be killed by the associated test suite. As a result, the misuse pattern cannot be detected. For other I/O related APIs, such as `InputStream()` or `File.open()`, missing method call `File.close()` is a common type of misuse. However, mutants representing such misuses can hardly be killed by the test suite. The reason is that such misuses are likely to induce performance issues. The test cases of client projects are rarely equipped with a proper oracle to identify such issues. Nonetheless, our empirical results (Section V-A) already confirm that a large number of real API misuse patterns can be detected via mutation analysis.

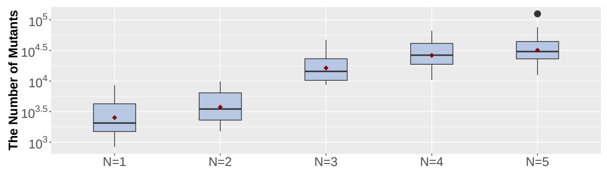


Fig. 12: The Number of Mutants Generated with Different N

C. *Threats to Validity*

The validity of our study is subject to the following threats. First, our study is limited to 73 Java APIs, and thus the generality to other APIs might be a threat. However, these APIs are popular (i.e., frequently discussed in Stack Overflow [4]) and systematically evaluated by existing studies [6]. Therefore, we believe that they are representative Java APIs. Performing experiments using more Java APIs is left as our future work. Second, our experiments involve only 16 client projects, and thus the results might not be generalizable to other projects. However, these projects are randomly selected based on the criteria described in Section IV-B. They are popular and from diverse domains. It is also worth mentioning that these projects only cover 55 of the 73 selected APIs. We did not purposely select more projects to cover the remaining 18 APIs in order to ensure the generality of our experiments.

VII. CONCLUSION AND FUTURE WORK

We present MUTAPI in this study, the first approach that leverages mutation analysis to discover API misuse patterns. It offers two major benefits compared with existing approaches. First, it does not require a large number of correct API usage examples. Therefore, it can be applied to detect misuse patterns of newly released APIs, whose number of usages might be limited, while existing approaches are less likely to work in such scenarios. Second, it goes beyond the simple assumption that a deviation from the most frequent patterns is a misuse. Existing approaches suffer from low precision due to such an assumption. We applied MUTAPI on 16 client projects with respect to 73 popular APIs. The results show that MUTAPI is able to detect substantial API misuse patterns. It also achieves a recall of 0.49 on the MUBENCH benchmark dataset [6], which significantly outperforms existing approaches.

In future, we plan to apply MUTAPI to those less frequently used APIs (e.g., those from newly released libraries) instead of popular ones to investigate whether MUTAPI can detect unknown API misuse patterns. We also plan to systematically investigate the effects of the test suite's quality on the effectiveness of our approach.

ACKNOWLEDGMENT

We thank Tianyi Zhang for his discussions toward this idea and anonymous reviewers for their constructive comments. This work was supported by the Hong Kong RGC/GRF Grant 16202917, the MSRA Collaborative Research Award, the GPU Grant Program by NVIDIA, the National Natural Science Foundation of China (Grant No. 61802164), the Science and Technology Innovation Committee Foundation of Shenzhen (Grant No. ZDSYS201703031748284) and the Program for University Key Laboratory of Guangdong Province (Grant No. 2017KSYS008). Zhendong Su was supported in part by United States NSF Grants 1528133 and 1618158.

References

[1] Y. Wang, M. Wen, Z. Liu, R. Wu, R. Wang, B. Yang, H. Yu, Z. Zhu, and S.-C. Cheung, "Do the dependency conflicts in my project matter?" in *Proceedings of the 2018 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018)*. ACM, 2018, pp. 1–12.

[2] "Maven repository," https://maven.apache.org/, 2018, accessed: 2018-02-28.

[3] U. Dekel and J. D. Herbsleb, "Improving api documentation usability with knowledge pushing," in *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 2009, pp. 320–330.

[4] T. Zhang, G. Upadhyaya, A. Reinhardt, H. Rajan, and M. Kim, "Are code examples on an online q&a forum reliable?: a study of api misuse on stack overflow," in *Proceedings of the 40th International Conference on Software Engineering*. ACM, 2018, pp. 886–896.

[5] L. Seonah, R. Wu, S.-C. Cheung, and S. Kang, "Automatic detection and update suggestion for outdated api names in documentation," *IEEE Transactions on Software Engineering*, 2019.

[6] S. Amann, H. A. Nguyen, S. Nadi, T. N. Nguyen, and M. Mezini, "A systematic evaluation of static api-misuse detectors," *IEEE Transactions on Software Engineering*, 2018.

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

[8] J. Sushine, J. D. Herbsleb, and J. Aldrich, "Searching the state space: a qualitative study of api protocol usability," in *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension*. IEEE Press, 2015, pp. 82–93.

[9] S. Fahl, M. Harbach, T. Muders, L. Baumgärtner, B. Freisleben, and M. Smith, "Why eve and mallory love android: An analysis of android ssl (in) security," in *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2012, pp. 50–61.

[10] 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.

[11] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov, "The most dangerous code in the world: validating ssl certificates in non-browser software," in *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2012, pp. 38–49.

[12] 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.

[13] C. Lindig, "Mining patterns and violations using concept analysis," in *The Art and Science of Analyzing Software Data*. Elsevier, 2016, pp. 17–38.

[14] 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.

[15] 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.

[16] 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 *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM, 2009, pp. 383–392.

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

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

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

[20] A. Wasylkowski and A. Zeller, "Mining temporal specifications from object usage," *Automated Software Engineering*, vol. 18, no. 3-4, pp. 263–292, 2011.

[21] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE transactions on software engineering*, vol. 37, no. 5, pp. 649–678, 2011.

[22] P. Ammann and J. Offutt, *Introduction to software testing*. Cambridge University Press, 2016.

[23] S. Hong, B. Lee, T. Kwak, Y. Jeon, B. Ko, Y. Kim, and M. Kim, "Mutation-based fault localization for real-world multilingual programs (t)," in *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2015, pp. 464–475.

[24] T. Loise, X. Devroey, G. Perrouin, M. Papadakis, and P. Heymans, "Towards security-aware mutation testing." in *ICST Workshops*, 2017, pp. 97–102.

[25] T. Mouelhi, Y. Le Traon, and B. Baudry, "Mutation analysis for security tests qualification," in *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION, 2007. TAICPART-MUTATION 2007*. IEEE, 2007, pp. 233–242.

[26] B. Baudry, F. Fleurey, and Y. Le Traon, "Improving test suites for efficient fault localization," in *Proceedings of the 28th international conference on Software engineering*. ACM, 2006, pp. 82–91.

[27] "Pit," http://pitest.org/quickstart/mutators/, 2018, accessed: 2018-02-28.

[28] R. Wu, H. Zhang, S.-C. Cheung, and S. Kim, "Crashlocator: locating crashing faults based on crash stacks," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ACM, 2014, pp. 204–214.

[29] W. Ying, W. Ming, W. Rongxin, L. Zhenwei, T. Shin Hwei, Z. Zhiliang, Y. Hai, and C. Shing-Chi, "Could I Have a Stack Trace to Examine the Dependency Conflict Issue?" in *Proceedings of the 41th International Conference on Software Engineering*. IEEE, 2019.

[30] I. Ahmed, C. Jensen, A. Groce, and P. E. McKenney, "Applying mutation analysis on kernel test suites: an experience report," in *IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2017, pp. 110–115.

[31] C. S. Timperley, S. Stepney, and C. Le Goues, "An investigation into the use of mutation analysis for automated program repair," in *International Symposium on Search Based Software Engineering*. Springer, 2017, pp. 99–114.

[32] M. Wen, J. Chen, R. Wu, D. Hao, and S.-C. Cheung, "Context-aware patch generation for better automated program repair," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: ACM, 2018, pp. 1–11. [Online]. Available: http://doi.acm.org/10.1145/3180155.3180233

[33] Y. Acar, M. Backes, S. Fahl, D. Kim, M. L. Mazurek, and C. Stransky, "You get where you're looking for: The impact of information sources on code security," in *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE, 2016, pp. 289–305.

[34] "Apache lang," https://github.com/apache/commons-lang.git, 2018, accessed: 2018-02-28.

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

[36] M. Linares-Vásquez, G. Bavota, M. Tufano, K. Moran, M. Di Penta, C. Vendome, C. Bernal-Cárdenas, and D. Poshyvanyk, "Enabling mutation testing for android apps," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017, pp. 233–244.

[37] K. Moran, M. Tufano, C. Bernal-Cárdenas, M. Linares-Vásquez, G. Bavota, C. Vendome, M. Di Penta, and D. Poshyvanyk, "Mdroid+: a mutation testing framework for android," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. ACM, 2018, pp. 33–36.

[38] R. Jabbarvand and S. Malek, "μdroid: an energy-aware mutation testing framework for android," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017, pp. 208–219.

[39] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, "Are mutants a valid substitute for real faults in software testing?" in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 654–665.

[40] "Apache category," https://projects.apache.org/projects.html?category, 2018, accessed: 2018-02-28.

[41] "Github topics," https://github.com/topics, 2018, accessed: 2018-02-28.

[42] "Apache fop," https://github.com/apache/fop.git, 2018, accessed: 2018-02-28.

[43] "Swingx," https://github.com/ebourg/swingx.git, 2018, accessed: 2018-02-28.

[44] "Jfreechart," https://github.com/jfree/jfreechart.git, 2018, accessed: 2018-02-28.

[45] "itextpdf," https://github.com/itext/itextpdf.git, 2018, accessed: 2018-02-28.

[46] "Apache math," https://github.com/apache/commons-math.git, 2018, accessed: 2018-02-28.

[47] "Apache text," https://github.com/apache/commons-text.git, 2018, accessed: 2018-02-28.

[48] "Apache bcel," https://github.com/apache/commons-bcel.git, 2018, accessed: 2018-02-28.

[49] "Fortress core," https://github.com/apache/directory-fortress-core.git, 2018, accessed: 2018-02-28.

[50] "Santuario-java," https://github.com/apache/santuario-java.git, 2018, accessed: 2018-02-28.

[51] "Pdfbox," https://github.com/apache/pdfbox.git, 2018, accessed: 2018-02-28.

[52] "Wildfly," https://github.com/wildfly-security/wildfly-elytron.git, 2018, accessed: 2018-02-28.

[53] "Apache jackrabbit," https://github.com/apache/jackrabbit.git, 2018, accessed: 2018-02-28.

[54] "H2database," https://github.com/h2database/h2database.git, 2018, accessed: 2018-02-28.

[55] "Curator," https://github.com/apache/curator.git, 2018, accessed: 2018-02-28.

[56] "Messagedigest," https://stackoverflow.com/questions/16133881/messagedigest-nosuchalgorithmexception, 2018, accessed: 2018-02-28.

[57] "Matcher group api," https://stackoverflow.com/questions/25851293/java-regex-why-matcher-group-does-not-work-without-matcher-find, 2018, accessed: 2018-02-28.

[58] J. Zhang, L. Zhang, M. Harman, D. Hao, Y. Jia, and L. Zhang, "Predictive mutation testing," *IEEE Transactions on Software Engineering*, 2018.

[59] B. Wang, Y. Xiong, Y. Shi, L. Zhang, and D. Hao, "Faster mutation analysis via equivalence modulo states," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2017, pp. 295–306.

[60] R. Paletov, P. Tsankov, V. Raychev, and M. Vechev, "Inferring crypto api rules from code changes," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2018, pp. 450–464.