Effective and Efficient API Misuse Detection via Exception Propagation and Search-Based Testing

Kechagia, Maria; Devroey, Xavier; Panichella, Annibale; Gousios, Georgios; van Deursen, Arie

DOI
10.1145/3293882.3330552

Publication date
2019

Document Version
Submitted manuscript

Published in
ISSTA 2019

Citation (APA)
Kechagia, M., Devroey, X., Panichella, A., Gousios, G., & van Deursen, A. (2019). Effective and Efficient API Misuse Detection via Exception Propagation and Search-Based Testing. In A. Moller, & D. Zhang (Eds.), ISSTA 2019 : Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (pp. 192-203). New York: Association for Computing Machinery (ACM). https://doi.org/10.1145/3293882.3330552

Effective and Efficient API Misuse Detection via Exception Propagation and Search-Based Testing

Maria Kechagia*
m.kechagia@ucl.ac.uk
University College London
London, United Kingdom

Xavier Devroey
x.d.m.devroey@tudelft.nl
Delft University of Technology
Delft, Netherlands

Annibale Panichella
a.panichella@tudelft.nl
Delft University of Technology
Delft, Netherlands

Georgios Gousios
g.gousios@tudelft.nl
Delft University of Technology
Delft, Netherlands

Arie van Deursen
arie.vandeursen@tudelft.nl
Delft University of Technology
Delft, Netherlands

ABSTRACT

Application Programming Interfaces (APIs) typically come with (implicit) usage constraints. The violations of these constraints (API misuses) can lead to software crashes. Even though there are several tools that can detect API misuses, most of them suffer from a very high rate of false positives. We introduce Catcher, a novel API misuse detection approach that combines static exception propagation analysis with automatic search-based test case generation to effectively and efficiently pinpoint crash-prone API misuses in client applications. We validate Catcher against 21 Java applications, targeting misuses of the Java platform's API. Our results indicate that Catcher is able to generate test cases that uncover 243 (unique) API misuses that result in crashes. Our empirical evaluation shows that Catcher can detect a large number of misuses (77 cases) that would remain undetected by the traditional coverage-based test case generator EvoSuite. Additionally, on average, Catcher is eight times faster than EvoSuite in generating test cases for the identified misuses. Finally, we find that the majority of the exceptions triggered by Catcher are unexpected to developers, i.e., not only unhandled in the source code but also not listed in the documentation of the client applications.

CCS CONCEPTS

• **Software and its engineering → Software libraries and repositories**; **Error handling and recovery**; **Software testing and debugging**; **Search-based software engineering**.

KEYWORDS

API misuse, software crash, static exception propagation, search-based software testing

*This work was done and completed at the Delft University of Technology.

ISSTA '19, July 15–19, 2019, Beijing, China

ACM ISBN 978-1-4503-6224-5/19/07…$15.00
https://doi.org/10.1145/3293882.3330552

ACM Reference Format:
Maria Kechagia, Xavier Devroey, Annibale Panichella, Georgios Gousios, and Arie van Deursen. 2019. Effective and Efficient API Misuse Detection via Exception Propagation and Search-Based Testing. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '19), July 15–19, 2019, Beijing, China.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3293882.3330552

1 INTRODUCTION

Developers use external libraries to increase the velocity and reduce the production cost of software projects [38]. While increasing productivity, this form of software reuse comes with several challenges: dependencies need to be kept up to date [15], developers must learn the intricacies of each imported Application Programming Interface (API), and resulting client programs should be robust, efficient, and responsive. Correctly using third-party APIs is not an easy task; many APIs are millions of lines of code large, interact with various external systems and, importantly, they use stacks of software that offer increasing levels of abstraction at the expense of observability of the workings of the underlying layers.

The fact that APIs are opaque to developers is known to lead to incorrect uses (or API *misuses* [3, 4]) since client applications can violate the (implicit) *usage constraints* (often referred to as *contract*) of those APIs. For example, a violation occurs when a client application calls a method that expects a non-null constrained formal parameter without validating (i.e., via null checks or error handling) the references used as arguments. API misuses can cause software reliability problems, originating from issues such as poor handling of user input and resource misuses [3], or even increasing the attack surface of client applications [18, 40]. Documentation is not adequate, as it is usually either outdated [10], defective [64], or just ignored by the developers of client applications [50].

While static API misuse detectors can successfully identify specific types of APIs misuses, they suffer from various limitations [4]. In particular, these approaches have a high rate of false positives, requiring developers to manually inspect (via cross-checking [4] or writing test cases [57]) and review large lists of candidate API misuses produced by static analysis. In fact, according to a recent empirical study of Johnson et al. [28], who interviewed developers, false positives and developer overload are the main sources of dissatisfaction with static analysis tools.

Dynamic analysis tools [21, 34, 35] can pinpoint crash-related bugs in the source code without any false positives. However, these approaches have to compromise between the exploration of the vast *search space* of possible execution paths of the application under test and the *time budget* allowed for the discovery of the bugs. If an API misuse requires an additional effort to get exposed (i.e., if only a few execution paths contain it), the analysis may fail to detect it. To overcome this, we can either set a larger search-time budget or reduce the search space by including only potentially interesting parts of the application under test.

The main idea of this paper is to restrict the search space of automatic test case generators to crash-prone API-call locations (*candidate misuses*), i.e., method calls that might throw exceptions at runtime. To this end, we define a novel approach, Catcher, that combines static exception propagation analysis and test case generation to effectively and efficiently discover candidate misuses in a software under test. Catcher can help developers by automating (i) the detection of misuses of the Java platform's API that can cause client application crashes and (ii) the generation of test cases triggering such crashes.

Catcher works as follows. First, static exception propagation [25, 51] (based on Soot [59]) identifies call paths that propagate runtime exceptions raised by API methods but remained unhandled, in the application *call sites*, potentially causing application crashes [19, 31]. The call sites to each one of those API methods represent candidate misuses, defining the search space for the test case generation. Then, traditional code coverage heuristics and the previously identified candidate misuses are used for focusing the automatic test suite generator EvoSuite [20, 21] towards the generation of test cases that trigger the candidates' (propagated) exceptions.

To evaluate our approach, we initially examine whether existing state-of-the-art test coverage-based approaches (here we consider EvoSuite) are effective and efficient in discovering API misuses (**RQ1**). Then, we assess whether the performance of automatic test suite generators, such as EvoSuite, on detecting crash-prone API misuses can be improved by Catcher (**RQ2**). Finally, we cross-check whether the exceptions that Catcher triggers are listed in the documentation (i.e., are expected) of the Java platform's API and client projects or not (**RQ3**).

We evaluate Catcher by using 21 Java client applications and targeting API misuses of the Java's JDK v. 1.8.0_181. We find that Catcher can automatically uncover 243 (unique) misuses of the Java platform's API in 21 client applications. The collected results show that Catcher revealed more API misuses (77 cases) that remained undetected by plain EvoSuite, while requiring less than 20% of the time.

In summary, we make the following contributions: (i) an investigation of state-of-the-art search-based test case generator (EvoSuite) for evaluating its efficiency and effectiveness on identifying API misuses in client projects; (ii) a novel technique (Catcher) that combines static exception propagation analysis and search-based software testing to maximize the number of found crash-prone API misuses in a software under test and minimize the time needed for discovering those misuses; (iii) an empirical evaluation involving 21 Java projects that shows the effectiveness and efficiency of the proposed solution. Finally, we provide the data of our study as well as the source code of Catcher and the scripts for the postprocessing of our results.[1]

```
class StringTokenizer implements Enumeration<Object> {
    ...
       /**
       * Returns the next token from this string tokenizer.
       * @return     ...
       * @exception  NoSuchElementException  if there are no
       *            more tokens in this tokenizer's string.
       */
334    public String nextToken() {
         ...
348      if (currentPosition >= maxPosition)
349          throw new NoSuchElementException();
       }
}
```

(a) `StringTokenizer` class from the Java JDK (API)

```
public class ZoneInfoCompiler {
    ...
687 private static class Rule {
      ...
696   Rule(StringTokenizer st) {
697     iName = st.nextToken().intern();
698     iFromYear = parseYear(st.nextToken(), 0);
699     iToYear = parseYear(st.nextToken(), iFromYear);
        ...
      }
    }
}
```

(b) `ZoneInfoCompiler` class from `joda-time` (client)

Figure 1: JDK API misuse in `joda-time`, issue #319

2 BACKGROUND

API misuses occur when a developer of a client application violates an implicit (or explicit) usage constraint of an API [4]. Figure 1 presents a misuse of the Java `StringTokenizer` API in `joda-time`:[2] the `nextToken` method might throw a `NoSuchElementException` if the condition on the current position in the input string is not satisfied (line 348). This post-condition is documented but not handled by the constructor of the `Rule` class in `joda-time`. The client (Figure 1(b)) neither performs any validation check for the input (*input sanitization*) before calling the API nor uses any *exception handling* mechanism for that API call. As a result, this API misuse propagates an exception from the Java platform's API to the caller, the `Rule` constructor, if the `st` parameter contains less than three tokens (line 696).

In their recent work, Amann et al. [4] proposed a classification of API misuses by identifying four *missing* and *redundant* API-*usage elements*: (i) missing (resp. redundant) *method calls* that should (resp. should not) be called before (resp. after) calling an API method; (ii) missing (resp. redundant) *conditions* that should (resp. should not) be checked before (resp. after) calling an API method; (iii) missing *iterations* for methods that should be called in a loop, checking a particular condition after each call, and redundant iterations for methods that should never be called in a loop; and (iv) missing (resp. redundant) *exception handling* that should (resp. should not) catch exceptions after calling an API method. According to this classification, the misuse in Figure 1(a) is both a missing condition misuse

[1] Available at https://github.com/mkechagia/Catcher.
[2] Also reported as a GitHub issue: https://github.com/JodaOrg/joda-time/pull/319.

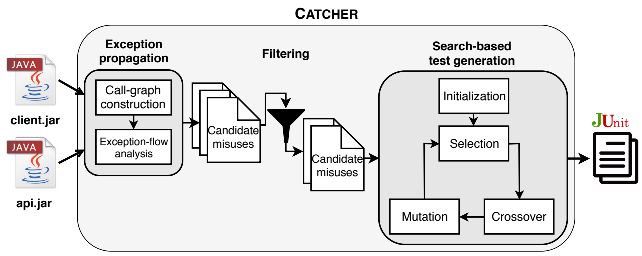

Figure 2: Catcher approach overview

and a missing exception handling misuse: while the documentation specifies that an exception might be thrown if the method is called when the input string to tokenize is exhausted, the client neither respects this implicit contract nor handle the thrown exception.

In this work, we look at *crash-related misuses* that can trigger exceptions propagated to the client, as they represent the vast majority of misuses [4]. Various static analysis methods are able to detect such kind of misuses, but they suffer from multiple limitations, including a high number of false positives, preventing their adoption in practice [4]. Moreover, even though some misuses can be effectively prevented through static analysis (e.g., by checking the presence of `try-catch` constructs to properly handle declared exceptions), other misuses, such as those related to input sanitization, require dynamic analysis (e.g., by using search-based testing) to cover the input and output domains of a method.

Various search-based test case generation approaches and tools have been proposed [20, 21, 35, 36] and shown effective in discovering real faults [2, 47, 54]. Those approaches rely on various kinds of criteria (like line and branch coverage [53], or input and output value domains coverage [52]) and algorithms (like genetic algorithms [22] or multi-objective algorithms [44, 45]) to explore a (*large*) search space, i.e., all the possible test cases that one could write for a given system under test. Furthermore, other approaches related to ours include C'n'C [16], which combines static checking and concrete test generation, and MutApi [37], which identifies API misuses based on mutation analysis. To the best of our knowledge, this work is the first that studies the relationship between search-based test case generation and crash-related API misuses.

3 THE CATCHER APPROACH

Catcher combines static exception propagation analysis [25, 51] with search-based test case generation [20, 21] to provide evidence of API misuses as a set of test cases. To achieve this, we use the approach described in Figure 2: (i) the static exception propagation analysis builds a call graph of the API under test with information about exceptions that might be thrown at runtime; (ii) using this graph, candidate misuses are identified by applying exception-flow analysis to API calls in the client, where such exceptions may be propagated; (iii) a rule set is then used to filter out propagated exceptions that are directly handled by the client (i.e., using `try-catch` constructs or `throws` clauses in method signatures); (iv) the remaining candidate misuses become coverage targets for the search-based test case generation. The test suite generated, in the latest step, will contain test cases that cover the target API calls in the client application and trigger the propagated exceptions.

The implementation of Catcher relies on Soot [59] for the call-graph construction and the exception-flow analysis, and on EvoSuite [20] for the search-based test case generation. We choose Soot for the following reasons: first, Soot is a well-known static analysis tool used in several research studies [8, 12, 25]; second, Soot's soundness and precision have been evaluated by researchers [49]; third, Soot can be easily used to analyze a Java program, by receiving only the application's `.jar` file as input; finally, the produced output can be easily used in exception-flow analysis [12, 25, 42]. After filtering the candidate misuses, Catcher relies on EvoSuite to focus the search-based test case generation process. We detail hereafter the configuration of EvoSuite, the modifications made on the standard implementation of the DynaMOSA algorithm, and the heuristics used for the test case generation.

3.1 Static Exception Propagation

3.1.1 Call-Graph Construction. The first step of our approach refers to the analysis of the source code of a software platform's API for spotting runtime (i.e., unchecked) exceptions that may be propagated to the callers of the API. This is done by building an annotated call graph, whose nodes represent the methods in the API and the edges denote the call dependency between each caller (*outgoing edge*) and callee (*ingoing edge*). The nodes are annotated with the list of runtime exceptions that might be thrown by the corresponding methods. Then, we build the annotated call graph for the client application under analysis and we connect it to the call graph of the API based on the method calls between the client and the API. On the resulting global call graph, we identify the first set of candidate misuses, which are the client nodes that have outgoing edges to the API nodes with annotated exceptions. Additional misuses are detected through the exception propagation analysis (see the next subsection 3.1.2).

For instance, the global call graph for the example in Figure 1 would contain two nodes: one is the method `nextToken()` from the API and the other one is the constructor `Rule` from the client. The two nodes are connected by an edge outgoing for the latter and ingoing for the former. Based on our analysis, the constructor `Rule` is a candidate misuse because it directly calls an API method that can throw an exception at runtime.

3.1.2 Exception-flow Analysis. To enlarge the set of candidate misuses, we use reachability analysis to propagate the exceptions from the API to the client. Specifically, for every node N_{api} of the API, its annotated exceptions are propagated backwards to all its adjacent nodes N_j (depth 1). Then the propagation is done for each node N_j recursively. The propagation path ends when the first client node in the global graph is encountered (depth k). All client nodes with exceptions propagated from API nodes are candidate misuses because they may expose exceptions thrown by the API.

The number of candidate misuses grows exponentially with the depth k that we consider. Buse and Weimer showed that exceptions with propagation depths larger than three are rarely listed in the documentation [12]. This possibly happens because it is not efficient for developers to consider these exceptions for debugging. For the sake of our study, focusing on the Java platform's API, we

consider calls with depth $k \leq 4$ to balance scalability and usability of CATCHER.

3.2 Filtering

The previous step identifies all API calls annotated with propagated exceptions. However, not all the identified calls are necessarily API misuses. In fact, API usage constraints can be satisfied at the client side through a combination of *programming language elements* [3]. A `try-catch` construct can be used to handle an exception propagated from the API to recover the client from the corresponding error state. Furthermore, the client application may include the propagated exception in the `throws` clause in the method signature of the caller. This postpones the exception handling to occur in other client methods and classes that exist later in the stack. These two types of programming language elements (i.e., `try-catch` constructs and `throws` clause in method signatures) can be easily identified via static analysis rules. To this aim, CATCHER uses a rule set to filter out API calls with propagated exceptions that are correct API usages: (i) calls made by the client to the API within a `try-catch` construct catching the propagated exception. Moreover, (ii) calls made by the client to the API within a method itself declaring a propagation of the exception using a `throws` clause. Besides, the rule set takes the Java `Exception` hierarchy into account. For instance, if an API method throws an `IOException` and the client has a `catch` clause for `Exception`, our rule set filters out the related candidate misuse. The list of the remaining candidate misuses is the input of the CATCHER's search-based test case generation.

3.3 Focused Search-based Test Generation

Given the list of candidate misuses identified in the previous steps, the generation of a test suite can be formulated as a search problem:

PROBLEM 1. *Let $M = \{m_1, \ldots, m_n\}$ be a set of candidate misuses (test targets) for a client class C. Our problem is to find a test suite $T = \{t_1, \ldots, t_n\}$ for C that identifies as many* API *misuses in M as possible by triggering the corresponding propagated exceptions.*

A candidate misuse $m_i \in M$ is successfully identified by a test case $t_j \in T$ if the following conditions hold: (1) t_j covers the candidate API call in the client class C (the call site), (2) t_j triggers the propagated exception, and (3) the last stack trace element in the crash stack trace is the (misused) API method. To solve the aforementioned problem, we need an adequate *heuristic* to guide a *search algorithm* toward covering the candidate misuses in M.

3.3.1 *Heuristic.*

We consider three state-of-the-art search heuristics. First, we use *line coverage*, which is defined as the sum of the *approach level* (*al*) and the normalized *branch distance* (*bd*) [36]: $bc_{m_i} = al(t_j, b_k) + norm\left(bd(t_j, b_k)\right)$ for test case t_j and branch b_k. Such a heuristic is widely applied in white-box testing and, in our case, it measures how far a test case t_j is to cover the API call site.

Additionally, we also consider *input coverage* $ic_{m_i}(t_j, b_k)$ and *output coverage* $oc_{m_i}(t_j, b_k)$ for the client method (caller) containing the potential API misuse. These two heuristics are black-box and aim to increase the input and the output data diversity during the test generation process. More diverse input/output can increase the likelihood of triggering unexpected behaviors [52], such as triggering the propagated exceptions.

For M, we have the following set of objectives to optimize:

$$\begin{cases} f(m_1) = min(bc_{m_1}), max(ic_{m_1}), max(oc_{m_1}) \\ \ldots \\ f(m_n) = min(bc_{m_n}), max(ic_{m_n}), max(oc_{m_n}) \end{cases}$$

3.3.2 *Search Algorithm.*

As in other search-based test case generation problems, covering as many candidate misuses as possible with CATCHER is a multi-target problem since a client class can contain multiple candidate misuses (targets) to be covered. Therefore, as a search algorithm, we choose the *Dynamic Many-Objective Sorting Algorithm* (DynaMOSA) [44], a state-of-the-art many-objective algorithm that optimizes multiple coverage targets, simultaneously. We opted for DynaMOSA since recent studies [13, 47] showed its better effectiveness and efficiency compared to other multi-target approaches, such as the *whole-suite approach*, *random search*, *evolution strategies*, and other many-objective algorithms.

In DynaMOSA, coverage targets (e.g., branches) correspond to search objectives, which are prioritized based on their structural dependencies in the control dependency graph of the class under test. The search starts by optimizing coverage targets positioned higher in the hierarchy; the other targets are incrementally reinserted in the search when their parent targets are satisfied (for instance, reach branch n before trying to reach branch $n + 1$). Using our heuristic, the algorithm executes as follows:

Initialization. The search starts by identifying a pool of client call sites (containing the candidate misuses) from M. Next, it generates a set of random test cases to produce an *initial population*.

Selection. To form the next generation, DynaMOSA applies *elitism* by using a *preference sorting* function [44]. For each candidate misuse m_i, the preference sorting function takes the test cases with the best individual objective scores and inserts them into the next population. The remaining test cases are then sorted and selected by the non-dominated sorting algorithm proposed in NSGA-II [17].

Reproduction. In each generation, parents are selected using the *tournament selection* and new test cases (offspring) are created by applying crossover and mutation operators [20].

Objective update. Once a test case reaches the API call site, the exception in m_i has to be thrown and propagated through the same methods. Thus, we ensure that the test t_j is archived to be part of the final test suite only if t_j triggers and propagates the exception through the same methods as m_i. When m_i is successfully detected, the list of objectives is (dynamically) updated by removing the corresponding objectives bc_{m_i}, ic_{m_i}, and oc_{m_i}.

Termination. The iteration process continues until M is covered or the search time is over.

4 EVALUATION PROTOCOL

4.1 Study Context

The context of our study consists of Java client applications and the third-party APIs they use. We selected the latest version of 21 open-source Java projects, whose names and characteristics are reported in Table 1. We chose these projects because they are well-known, regularly maintained, and have been already used in the related literature to assess the performance of testing tools (e.g., [44, 45]) or to build datasets of known bugs (e.g., [29]). Also, they have different sizes, development teams, and application domains (e.g., byte code

Table 1: Java projects in our Benchmark

ID	Name	Version	Files	LOC	# Candid. Misuses
BCEL	bcel	6.2	489	39K	223
CLI	commons-cli	1.4	50	7K	114
CODEC	commons-codec	1.12	124	20K	580
COLL	commons-collections	4.2	535	63K	213
COMP	commons-compress	1.17	352	43K	587
LANG	commons-lang	3.7	323	76K	1024
MATH	commons-math	3.6.1	1617	209K	411
EASY	easymock	3.6	204	14K	217
GSON	gson	2.8.5	206	25K	157
HAMC	hamcrest-core	1.3	152	7K	34
JACK	jackson-databind	2.9.6	919	114K	251
JAVS	javassist	3.23.1	527	82K	313
JCOM	jcommander	1.71	139	6K	68
JFCH	jfreechart	1.5.0	990	134K	681
JODA	joda-time	2.10	330	86K	1053
JOPT	jopt-simple	5.0.4	192	9K	107
NATT	natty	0.13	27	3K	33
NEO4	neo4j-java-driver	1.6.2	510	52K	784
SHIRO	shiro-core	1.3.2	653	31K	223
XJOB	xwiki-commons-job	10.6	67	3K	616
XTEX	xwiki-commons-text	10.6	3	101	11

manipulation, math library, command line parser), forming a well-diversified benchmark. Their source code and documentation are publicly available for the reproducibility of our results.

The projects in our benchmark use various third-party APIs. However, we consider only the APIs of the Java JDK (version 1.8.0_181) rather than all possible third-party APIs used by these projects. We do this guided by the fact that the Java platform's API is extensively documented, more than other third-party APIs [30] (e.g., Android, Apache commons). Having well-documented APIs is critical for us to classify and understand the detected misuses (see Section 5). Furthermore, the Java platform is well-known and millions of developers use it to build their programs. While in our study we focus on the Java platform's API, our approach can be applied to verify the usages of other third-party APIs, including those that are not as extensively documented as the Java API.

4.2 Research Questions

We investigate the following research questions:

RQ1: *How do existing unit level coverage-based test generation tools perform in discovering API misuses?*

With this first research question, we aim to examine the effectiveness and efficiency of existing state-of-the-art automatic test suite generators regarding their capability to generate test cases able to expose API misuses. We are interested in investigating this research question because API calls are statements in the source code of the client applications and they can be covered by traditional unit-test generation tools (such as EvoSuite) tailored to maximize coverage-based criteria (e.g., branch coverage). However, covering API call sites does not necessarily imply that the corresponding tests can trigger the exceptions propagated from the APIs, exposing the misuses. To the best of our knowledge, we are the first to apply and evaluate such tools to study how they help to expose API misuses.

RQ2: *Does* Catcher *improve the performance of existing test coverage-based approaches on detecting* API *misuses?*

With this second research question, we investigate the impact of reducing the search space in test case generation by using information from the static exception propagation. Namely, we examine whether we can get test cases that expose more API misuses, and in less time, by considering only particular paths with candidate misuses identified by Catcher during the search. To this extent, we compare the effectiveness and efficiency of Catcher and EvoSuite for API misuses detection.

RQ3: *What types of* API *misuses does* Catcher *expose?*

Using Catcher, we can argue about particular API misuses (related to constraint misuses and exception handling misuses) at the code level. APIs also come with their reference documentation, which can significantly affect the robustness of client applications [12, 14, 31]. If an exception that might throw at runtime is not listed in the documentation (i.e., it is unexpected), developers stay unaware of the possible manifestation of that exception at runtime. Then, developers usually leave these exceptions unhandled decreasing the robustness of their programs. Based on that, we want to examine whether the exceptions triggered by Catcher are documented (and therefore expected) or not.

4.3 Baseline Selection and Parameter Setting

To answer **RQ1**, we select EvoSuite [20, 21] as our baseline. EvoSuite is a state-of-the-art testing framework for generating unit test suites for Java classes. It won the latest two editions of the SBST tool context [39, 48], which showed its ability to produce tests with higher code coverage and better fault detection capability compared to alternative tools (e.g., Randoop [43]). EvoSuite implements various search algorithms for test case generation. In our study, we use the Dynamic Many-objective Sorting Algorithm (DynaMOSA) proposed by Panichella at al. [44], which is the same many-objective genetic algorithm used in Catcher. We select DynaMOSA because it outperforms other multi-target and single-target approaches as demonstrated by recent studies [13, 44, 45, 47] that compare different algorithms in test case generation.

EvoSuite optimizes eight test criteria simultaneously as they are described by Rojas et al. [52]: *branch*, *line*, *weak mutation*, *input*, *output*, *method*, and *exception* coverage. In this study, we consider all these criteria as recent studies showed that their combination increases the fault detection capability of the generated test suites [27, 46]. Suites with higher fault detection capability are likely able to detect more crash-related API misuses. When enabled with *exception* coverage, EvoSuite archives all test cases that (i) trigger an exception and (ii) are created when trying to maximize the other aforementioned coverage criteria. The archived test cases are included in the final test suite, which contains test cases that allow reaching high code coverage plus all test cases generated during the search that trigger an exception. Some of these crashes might be related to propagated exceptions due to API misuses.

To answer **RQ2**, we compare EvoSuite and Catcher. Both EvoSuite and Catcher share the same search algorithm (i.e., DynaMOSA) and the same test case generation engine (e.g., genetic operators, chromosome representation). The differences between the DynaMOSA algorithm in EvoSuite and Catcher regard the objectives they optimize. The former targets all source code elements (e.g., branches, lines) for code coverage optimization. Instead,

as explained in Section 3.3, the latter targets only candidate APIs misuses (that are specific lines in the source code) as well as input and output coverage for the client methods (callers) containing candidate misuses.

Parameter Setting. Search algorithms have various parameters to set, which may potentially impact the results of our study. However, Arcuri and Fraser [7] showed that parameter tuning in search-based software engineering is extremely expensive and does not provide substantial improvements compared to default parameter values. We use the default parameter values suggested in the literature [7, 21, 45]: EvoSuite and Catcher were configured with a population size of 50 test cases; test cases are selected using the *tournament selection*, with tournament size s=10. Each newly generated test t is mutated through a *uniform mutation* with the probability p_m=1/n, where n is the number of statements in t. EvoSuite and Catcher were configured with a search budget of three minutes per each class under test. We use this setting because it represents a *reasonable* compromise between running time and coverage as reported in the related literature [22, 47].

4.4 Experimental Protocol

The number of candidate misuses for each project in our benchmark is reported in the rightmost column of Table 1. Then, test cases are generated only for classes that, according to the first two steps in Catcher, contain candidate API misuses. Therefore, classes with no candidate misuses are not targeted by Catcher during the test case generation phase. Instead, EvoSuite does not identify candidate API misuses before starting the test case generation process.

To address the random nature of EvoSuite and Catcher, we ran each tool 25 times on each class under test. For EvoSuite, the classes under test are all the classes in the benchmark projects. For Catcher, the classes under test are only those classes identified as having candidate API misuses. In total, for Catcher, we performed 905 (classes) × 25 (repetitions) ≈ 22,625 search executions, with three minutes of search budget per each execution. For EvoSuite, the number of classes increases to 8,409 corresponding to ≈ 210,225 search executions, with three minutes of search budget each. All executions were performed on a two node cluster. Each cluster node ran a GNU/Linux system (Ubuntu 16.04 LTS) with Linux kernel 4.4.0, on a dual 8-core 2.4GHz Intel E5-2630v3 CPUs with 64GB of RAM. We used Oracle's Java VM (JVM) version 1.8.0_181, allocating up to 12GB for the JVM.

In each run, we collected the generated test suite and the total running time needed for completing the search. We use the collected data to answers **RQ1** and **RQ2**. In particular, we re-executed the generated test suites (by Catcher and EvoSuite) at the end of each search, to identify test cases that triggered an exception. Then, we compared the corresponding crash stack traces with the list of candidate API misuses identified with the exception propagation analysis. Catcher and EvoSuite expose a target misuse m_i if they generate a test case t_j that triggers an exception propagating from the API to the client in the same way as m_i. In other words, the detection requires that the following two conditions hold: (i) the name of the exception triggered by t_j and the name of the propagated exception m_i coincide; (ii) the chain of call sites of m_i appears in the stack trace of the exception triggered by t_j.

To evaluate the effectiveness in **RQ1** and **RQ2**, we compute the *number of misuses exposed* by Catcher and EvoSuite in each independent run. To measure the efficiency, we compute the *total execution time* taken by Catcher and EvoSuite for each project in each independent run. The running time for Catcher is measured by taking into account (1) the time required by the static exception propagation analysis (for all steps in Section 3.1) to identify potential API misuses, (2) the test case generation time (i.e., up to three minutes) and (3) the post-processing. For EvoSuite, the running time includes (1) the search budget (up to three minutes) and (2) the post-processing. More specifically, we post-processed the test suites generated by the two tools to remove statements in the test cases that do not contribute to coverage or trigger the exceptions (test suite minimization); furthermore, assertions are automatically generated using mutation analysis [24]. Notice that Catcher uses the post-processing engine of EvoSuite.

We compare both approaches (Catcher and EvoSuite) by considering the median and the interquartile range (IQR) of the number of exposed API misuses and the running time over 25 independent repetitions. Due to space limitation, the results are reported at the project level. We use the non-parametric Wilcoxon Rank Sum test with a confidence level $\alpha = 0.05$ to assess the statistical significance of the differences (if any). Besides, we use the Vargha-Delaney $\hat{A}_{12}$ statistic [60] to measure the effect size of such differences.

To address **RQ3**, we inspect the API misuses detected by Catcher by analyzing and re-executing the generated tests, and inspecting the source code and the documentation (Javadoc) of both the API callers in the client applications and the misused APIs themselves. To reduce biases, we partially automated the analysis using a script which checks whether the propagated exceptions were adequately documented (e.g., reported in the Javadoc with the `@throws` or `@exception` tags) (i) in the APIs of the Java JDK and (ii) in the documentation or the source code comments of the callers (call sites) in client applications. The output of this analysis resulted in a classification of three types of API misuses that are discussed at the end of Section 5.

5 RESULTS

Results of RQ1. Table 2 reports, for each project, the median, the interquartile range (IQR), and the total number of unique API misuses detected by EvoSuite across 25 runs. EvoSuite can detect, on average, 123 crash-related API misuses in the 21 benchmark projects. If we consider all API misuses that are detected at least once across 25 runs, the total number of detected misuses is 166. While EvoSuite can detect some misuses by maximizing code coverage, the variability of the results is very high for some projects. For example, if we consider the project `Gson`, we notice that EvoSuite detects on average two misuses. However, if we run EvoSuite multiple times, the total number of unique misuses being detected is eight. Therefore, the set of discovered misuses differs substantially between two independent runs. To have more reliable results, we would need to run EvoSuite multiple times, with a corresponding increment of the overall running time. A similar observation can be done for other projects, such as `apache-commons-compress` (COMP), `jackson-databind` (JACK), `JFreeChart` (JFCH), and `joda-time` (JODA).

Table 2: Statistics on the comparison between the number of crash-related API misuses exposed by CATCHER and EVOSUITE.

	Catcher			EvoSuite			Significance	
Project	Median	IQR	Total	Median	IQR	Total	p-value	$\hat{A}_{12}$
BCEL	8	1.5	9	7	1	7	**<0.0001**	**0.87 (L)**
CLI	0	-	0	0	-	0	1.0000	–
CODEC	9	-	9	9	1	9	**<0.0004**	**0.72 (M)**
COLL	10	-	10	6	2	9	**<0.0001**	**1.00 (L)**
COMP	28	7	34	10	3	18	**<0.0001**	**0.99 (L)**
LANG	30	1	32	20	3.75	23	**0.0052**	**0.83 (L)**
MATH	10	2	12	9	-	9	**<0.0001**	**0.88 (L)**
EASY	11	1.25	16	9	1.75	11	**<0.0001**	**0.96 (L)**
GSOM	12	4	16	2	6	8	**<0.0001**	**1.00 (L)**
HAMC	0	-	0	0	-	0	1.0000	-
JACK	5	1	6	2	-	5	**<0.0001**	**0.95 (L)**
JAVS	4	-	4	2	-	2	**<0.0001**	**1.00 (L)**
JCOM	1	-	2	1	-	1	0.0810	0.56
JFCH	21	5	27	14	3	23	**<0.0001**	**0.99 (L)**
JODA	20	1	25	5	1	11	**<0.0001**	**1.00 (L)**
JOPT	3	-	3	3	-	3	1.0000	0.50
NATT	4	-	4	3	-	4	**<0.0001**	**0.96 (L)**
NEO4	14	1	17	8	-	8	**0.0219**	**0.91 (L)**
SHIRO	7	-	7	6	-	6	**<0.0001**	**1.00 (L)**
XJOB	10	0.75	10	7	3	9	**<0.0001**	**0.96 (L)**
XTEX	0	-	0	0	-	0	1.0000	0.50
Total	207		243	123		166		

Table 3: Execution time (in *s*) for CATCHER and EVOSUITE.

	Catcher				EvoSuite		Catcher
	Exc.	Search			Search		is %
Project	Prop.	Median	IQR	Total	Median	IQR	faster
BCEL	621	5,430	2	6,051	93,879	783	94%
CLI	318	2,353	5	2,671	5,475	21	52%
CODEC	327	6,878	3	1,014	17,418	41	95%
COLL	348	6,154	2	6,502	117,995	508	95%
COMP	339	13,780	17	14,119	87,029	400	86%
LANG	343	12,552	72	12,895	30,634	196	58%
MATH	593	12,252	162	12,845	91,825	456	87%
EASY	642	8,879	6	9,521	152,679	533	94%
GSON	333	3,105	181	3,438	17,775	52	81%
HAMC	306	1,086	-	1,392	8,598	7	84%
JACK	371	8,636	1,203	9,007	123,999	698	93%
JAVS	587	9,786	222	10,373	58,331	234	83%
JCOM	317	2,353	-	2,670	14,228	51	82%
JFCH	633	26,488	113	27,121	72,139	319	63%
JODA	540	5,977	175	6,517	46,007	220	86%
JOPT	314	3,085	2	3,399	12,294	9	73%
NATT	759	1,267	-	2,026	45,314	6	96%
NEO4	542	13,972	1,739	13,983	157,081	803	88%
SHIRO	582	5,979	1	6,561	71,270	129	91%
XJOB	591	5,558	158	6,149	100,829	222	94%
XTEXT	299	362	-	661	862	0.33	24%
Total	9,716	149,741		159,457 (~ 2 *days*)	1,327,080 (~ 15 *days*)		81% (mean)

This variability is due to the fact that, in each generation, EVOSUITE focuses the search on the uncovered targets (e.g., branches and mutants). Indeed, as soon as a new coverage target b is covered, the corresponding test case is stored in the final test suite, and b is removed from the set of objectives to optimize [44, 53]. While this heuristic has been proven to lead to a higher overall coverage [44, 53], it is not suitable for detecting API misuses. Covering the API call site is a *necessity* but *not a sufficient* condition to expose the API misuses and trigger the propagated exception.

Concerning running time, EVOSUITE requires on average 17 hours to complete the test generation for one project. The overall running time is proportional to the number and the complexity of classes in the project under test. Indeed, it varies from 14 minutes (`xwiki-text` has three classes) to 1 day, 19 hours, and 38 minutes (for `EasyMock`) on average. This highlights the need for test case generation approaches that focus on API misuses.

Results of RQ2. Table 2 shows that CATCHER detects on average 84 (+68%) API misuses compared to EVOSUITE across the 25 runs. In total, the number of unique API misuses detected by CATCHER in all runs is 243, i.e., +77 unique misuses over EVOSUITE. These differences are also confirmed by the statistical analysis reported on the right side of Table 2: for 16 projects out of 21 (76%), CATCHER identifies significantly more API misuses than EVOSUITE. The effect size is always *large* (in 15 projects out of 16) and *medium* (in one project).

Let us consider the example in Figure 3 of an API misuse detected by CATCHER but not by EVOSUITE for the class `KthSelector` from the project `apache-commons-math`. The `ArrayIndexOutOfBoundsException` is thrown at line 120 of the API `Arrays.rangeCheck` and propagated back to the client application in the method `select`. An excerpt of the code of this method is reported in Figure 3-(a) while Figure 3-(c) reports the test case generated by CATCHER. When executing the test case, the client method `select` invokes the method `Arrays.sort` using as parameters an array of size 17 and the variables `begin`=12 and `end`=19. The value of the variable `end` is larger than the size of the array and, thus, the exception is thrown in `Arrays.rangeCheck`, which is indirectly invoked. Notice that the value of these two variables is computed within the `while` loop in lines 84–113. This example is an API misuse because the client method `select` should validate the input data (e.g., the length of the array) before invoking the API.

Table 4 reports the number of API misuses detected by both approaches, CATCHER and EVOSUITE, as well as the number of misuses detected by one approach (e.g., CATCHER) but not by the other one (e.g., EVOSUITE). We observe that 163 unique API misuses are detected by both approaches; 80 unique API misuses are detected only by CATCHER; two unique misuses are detected only by EVOSUITE. Through manual investigation, we discovered that these two misuses are detected by EVOSUITE thanks to the *weak mutation* coverage, which leads to generating input data able to *weakly* kill mutants (infection state). This input data might increase the likelihood of exposing misuses, although it happens for only two cases in our benchmark. Future work will be devoted to investigating other coverage criteria in CATCHER, including the *weak mutation* coverage.

Finally, Table 3 reports the running time of CATCHER and EVOSUITE. CATCHER requires less than 20% of the time spent by EVOSUITE in total. On a per project basis, CATCHER is on average 80% faster, with a maximum speedup of 96% for project `natty`. The smallest difference is observed in the case of `xtext`: as `xtext` is a very small library comprising only 100 LOC, the setup cost for the static analysis phase in CATCHER dominates the total execution time. For all projects, the differences are statistically significant according to the Wilcoxon test (p-values<0.0001) with a *large* effect size ($\hat{A}_{12} > 0.90$).

Results of RQ3. Based on the results of our qualitative analysis (whose procedure is discussed in Section 4.4), we identified

Table 4: Overlap between CATCHER and EVOSUITE regarding the unique crash-related API misuses detected across 25 repetitions.

Project	Catcher ⋂ EvoSuite	Catcher \ EvoSuite	EvoSuite \ Catcher
BCEL	7	2	-
CODEC	9	-	-
COLL	9	1	-
COMP	16	18	1
LANG	23	9	-
MATH	8	4	1
EASY	11	5	-
GSON	8	8	-
JACK	5	1	-
JAVS	2	2	-
JCOM	1	1	-
JFCH	23	4	-
JODA	11	14	-
JOPT	3	-	-
NATT	4	-	-
NEO4	8	9	-
SHIRO	6	1	-
XJOB	9	1	-
Total	163	80	2

```
public double select(final double[] work, final int[] pivotsHeap, final int k) {
 80       int begin = 0;
 81       int end = work.length;
          ...
 84       while (end - begin > MIN_SELECT_SIZE) {
          ...
          }
113       Arrays.sort(work, begin, end);
114       return work[k];
}
```

(a) Class `KthSelector` from `apache-commons-math`

```
java.lang.ArrayIndexOutOfBoundsException:
        at java.util.Arrays.rangeCheck(Arrays.java:120)
        at java.util.Arrays.sort(Arrays.java:440)
        at org.apache.commons.math3.util.KthSelector.select(KthSelector.java:113)
```

(b) ArrayIndexOutOfBoundsException occurs in the method `select`

```
@Test
public void test12()  throws Throwable  {
      double[] doubleArray0 = new double[17];
      KthSelector kthSelector0 = new KthSelector();
      int[] intArray0 = new int[3];
      intArray0[0] = 19; intArray0[1] = 11; intArray0[2] = 15;
      kthSelector0.select(doubleArray0, intArray0, 15);
}
```

(c) The test that is generated by Catcher

Figure 3: Example of API misuse detected by CATCHER but not by EVOSUITE.

three types of misalignment between the API reference documentation and the API usages in the client applications. The numbers of misuses for each type are reported in Table 5.

Type#1. The first type of misuses is *Complete API documentation—Inconsistent client.* This category includes propagated exceptions that are listed in the documentation of an API method. However, these exceptions are neither handled in the caller methods, e.g., via check conditions or `try-catch` constructs to handle the raised (yet documented) exceptions, nor documented in the Javadoc of the client application. We found that the large majority (82%) of misuses exposed (triggered) by CATCHER falls in this category. For one project, `neo4j-java-driver`, we also submitted and received confirmation by developers for such relevant identified issues, which were actually fixed.[3] This result highlights the practical usefulness of automated tools, such as CATCHER, to notify developers of client applications about possible misuses of an API method.

Table 5: Categories of triggered crashes per project.

Project	Type#1	Type#2	Type#3
BCEL	9	0	0
CODEC	4	2	3
COLL	8	2	0
COMP	23	5	6
LANG	25	4	3
MATH	9	1	2
EASY	16	0	0
GSON	10	0	6
JACK	4	2	0
JAVS	4	0	0
JCOM	1	1	0
JFCH	26	1	3
JODA	21	4	0
JOPT	3	0	0
NATT	4	0	0
NEO4	17	0	0
SHIRO	6	1	0
XJOB	9	1	0
Total	199	24	20

Type#2. The second type of misuses is *Incomplete API documentation—Unaware client.* This category includes propagated exceptions that are not listed in the API reference documentation of the APIS and possibly this leads developers of client applications to API misuses. From our analysis, we discovered that around 10% of the detected misuses falls in this category. This is in line with empirical studies that argue about the impact of undocumented exceptions on applications' robustness [14, 31].

Type#3. The third type of misuses refers to the *Complete API documentation—Consistent client.* This category includes propagated exceptions that are listed in the documentation of an API method, but the client chooses explicitly not to handle them in the source code. Consider the following scenario. The developers of a client application are aware of a propagated exception and list this in the application's documentation. A generated test exercises an expected behavior of the client method. Nevertheless, the API misuse remains in the source code and can lead to crashes. Then, the generated test can still be added to the existing test suite of the client application and can be used in later regression testing activities. From our investigation, we found that around 8% of the detected misuses falls in this category.

6 DISCUSSION

Research implications. The findings of **RQ2** indicate that focused testing of API uses is not equivalent to merely maximizing traditional coverage criteria, although candidate misuses are statements in the source code of the client applications. The better detection capability and performance of CATCHER compared to plain EVOSUITE is due to the heuristics (focusing the search-based test case

[3] https://github.com/neo4j/neo4j-java-driver/issues/520

generation) of Catcher. We hope that our study draws new research directions towards the evaluation of the benefits of focused search-based test case generation. New studies could possibly consider the detection of different types of API misuses e.g., ones that are related to security and energy efficiency issues.

It is worth noting that EvoSuite combined with the static exception propagation analysis as implemented in Catcher is more effective and efficient than using only its default criteria. In particular, in Catcher, we run EvoSuite by targeting only the classes for which static analysis provided a list of (filtered) candidate misuses and considering only a subset of the coverage targets, i.e., API call sites, input, and output coverage for the caller methods. Furthermore, the exception propagation chains generated by the static analysis help us to automate the *test oracle*, i.e., to discover which generated tests can detect the misuses. In the traditional setting, plain EvoSuite should target all the classes of the examined projects, resulting in thousands of test suites (with multiple test cases each) that should be manually evaluated by developers [21] to identify those cases able to trigger the misuses. Therefore, we opt for more studies that combine the strengths of static analysis and automated search-based test case generation.

Practical implications. Catcher can identify API misuses in client programs. Both developers of client programs and developers of APIs can use this information to make their programs more robust.

Developers of client programs can use Catcher to correctly identify, handle and recover runtime errors caused by combinations of wrong inputs. The test cases that Catcher generates can be used as a safety net against regressions in both the client and the API code. The runtime cost of Catcher makes it suitable for use in release pipelines: Catcher could in less than 4 hours examine a 250k LOC program (for `commons-math`). As part of an automated release process, Catcher could be proven useful to identify last minute issues. Finally, Catcher could be further improved to examine changes on a per commit basis: this would allow it to run as part of continuous integration pipelines, in order to support interactive quality assurance processes, such as code review.

Moreover, the information that Catcher produces can be used upstream by developers of APIs, to help them improve the robustness of error-prone methods against wrong or adversary inputs. API developers can make their code less susceptible to runtime exceptions by guarding against inputs that Catcher identifies as erroneous. Catcher tests encode an implicit invocation protocol. API developers can inspect such tests to identify and fix initialization or ordering issues that may lead their APIs to fail. If corrective action is not possible, documentation can be used to make invocation protocols explicit to clients.

7 THREATS TO VALIDITY

Internal validity. State-of-the-art mining and static analysis detectors for API misuses suffer from low precision [4]. Instead, our results show that Catcher can precisely detect actual misuses and provide empirical evidence of such misuses through generated test cases. However, we acknowledge that we cannot make any claim about the completeness of Catcher because there is no ground truth for the projects in our benchmark. Further investigation in that regards is part of our future agenda. Furthermore, the automated analysis we performed to evaluate whether the exceptions triggered by Catcher are listed in the documentation of the Java platform's API and the projects can also suffer from imprecision. Even though we have also manually inspected and confirmed the results, maybe a few exceptions found to be as undocumented could finally be listed in the documentation. Another potential threat to internal validity is the randomized nature of the genetic algorithms and the seeding-based random search. To address this threat, we followed the guidelines from the related literature[6]: we launched each algorithm 25 times, and we used sound statistical tests, namely the Vargha-Delaney $\hat{A}_{12}$ statistic and the Wilcoxon Rank Sum test, to draw any conclusion. Another threat is related to the parameter setting of the search algorithms. We used the parameter values suggested by the related literature [7, 45, 55].

External validity. We acknowledge that our results regard one particular API, i.e., the Java 8 platform's API. Future work includes the analysis of other Java API versions, as well as additional third-party Java libraries used by client applications. Additionally, even though our results are related to specific client applications, we used a large benchmark of well-diversified and well-known software projects. Thus, we expect that our main conclusions can also apply to other benchmarks.

Reliability validity. For the reproducibility of our study, we have made the source code (Catcher), the processing scripts, and our data publicly available.[4] Specifically, in the data, we include the examined projects and APIs (input), as well as the found test cases (output).

8 RELATED WORK

Static API misuse detection. To assist developers to detect API misuses, researchers have proposed tools that leverage techniques including mining of software repositories and static analysis [1, 33, 41, 57, 58, 61]. In general, these tools work as follows. Start by mining correct API usage patterns, from existing code bases, and continue by classifying infrequent patterns observed in target projects as candidate misuses. The produced candidates should be then manually reviewed by developers. The available techniques mainly differ on: (i) the representation (e.g., via graphs [41, 61], formal concepts [33]) of the method call usages and (ii) the mining algorithms (e.g., frequent-itemset mining [58], model checking [1], frequent-subgraph mining [41]) used to detect infrequent patterns (outliers)—based on thresholds defined *a priori*.

Recently, Amann et al. [4] compared 12 state-of-the-art misuse detectors on a set of known APIs misuses collected from existing bug datasets. They found that all detectors suffer from a number of limitations. Initially, all detectors have low precision (below 12%) as they produce a large number of false positives. This means that, on average, the tools report less than 1.5 API misuses in the top-20 of their results. Yet these tools typically produce an extensive list of candidate APIs misuses, which developers have to manually check and approve. Also, most tools require large code bases to distinguish uncommon—but correct usages—from actual misuses.

[4] Available at https://github.com/mkechagia/Catcher

Contrary to previous approaches, CATCHER (i) effectively and efficiently identifies crash-related API misuses eliminating the need for the manual assessment of the candidate API misuses produced by the static analysis—it applies filtering and search-based test case generation to validate these candidates automatically; (ii) it does not need a code base for learning to distinguish good from bad API usage patterns—it uses exception propagation analysis to automatically pinpoint candidate API misuses. In essence, CATCHER sacrifices recall (it does not report all possible misuses) for achieving high precision (all reported misuses are indeed misuses).

Static exception propagation. Several tools exist for statically identifying possible exceptions that a method can throw at runtime [56]. Robillard and Murphy implemented Jex that applies interprocedural analysis and finds all the exception types that a specific method of a Java program can generate at runtime [51]. Vallée-Rai et al. developed the SOOT Java byte code optimization framework that can identify might-thrown exceptions for API methods, using (in the first instance) intra-procedural static analysis [59]. Fu and Ryder presented an inter-procedural exception-flow analysis technique, based on SOOT, for the examination of the exception handling architecture of software systems [25]. Bravenboer and Smaragdakis combined inter-procedural exception-flow analysis and points-to analysis for better precision in call-graph construction [11]. Garcia and Cacho introduced an inter-procedural exception-flow analysis tool (eFlowMining) for .NET, which visualizes error handling constructions [26].

The exception propagation of CATCHER is inter-procedural and mainly differs from peer approaches in the filtering (Section 3.2) of the found API-misuse candidates coming from the initial stage of the static exception propagation analysis (Section 3.1). Our filtering approach helps us to keep in the candidate misuses' list only the real API misuses that refer to: (i) uncaught exceptions and (ii) undeclared exceptions in the `throws` clause of the method signature of a caller.

Search-based software testing. Most of the research effort in search-based software testing (SBST) has been devoted to three main aspects: (i) evaluating fault detection capability of generated tests (e.g., [23, 27]), (ii) defining heuristics to guide the search process (e.g., [32, 62]), (iii) designing and evaluating different search algorithms (e.g., [5, 44, 45]). The goal of SBST tools consists in generating test cases/suites maximizing some coverage criteria (e.g., *branch*, *line*, and *statement* coverage) [21]. Recent studies [27, 52] empirically investigated the effect of combining multiple coverage criteria on the quality of the generated test suites and showed a positive impact on the fault detection capability.

SBST techniques use heuristics that are specific to each coverage criterion and measure how far a candidate test case/suite is from covering each coverage target (e.g., branches). For example, common heuristics for branch coverage include the *branch distance* [32] and the *approach level* [62].

These heuristics are then used to guide search algorithms towards generating tests with higher coverage. The earliest search strategy is the single-target approach, which attempts to satisfy one coverage target (e.g., one branch) at a time through multiple re-executions of the search (e.g., genetic algorithms). More recent approaches [20, 21, 44, 45] handle all coverage targets (e.g., all branches) at once with one single execution of the search. Rojas at al. [53] showed that multi-target approaches are superior to the single-target ones, while Panichella at al. [44, 45] demonstrated the higher capability of many-objective search compared to alternative multi-target approaches in reaching higher code coverage.

Compared to the advances of SBST mentioned above, in this paper, we use exception propagation to identify candidate API misuses in the source code of client applications. Then, we use both coverage-based heuristics to guide a many-objective search toward covering the identified API call sites and expose the propagated exceptions. Therefore, compared to existing techniques, our approach focuses the search on the candidate API misuses rather than targeting all code elements (e.g., branches) of client applications.

Hybrid approaches. Several hybrid (static and dynamic) analysis approaches have been developed in the past for software verification. For instance, Babić et al. used static analysis to guide their symbolic-execution based automated test generation tool to identify vulnerabilities [9]. Additionally, Zhang et al. combined static and dynamic automated test generation approach to identify bugs related to the sequence of method calls among the classes of a Java project [63]. Also, Ma et al. developed a hybrid technique that uses static analysis to extract knowledge from a project under test to guide the run-time test generation [34]. Finally, Csallner and Smaragdakis developed C'n'C that automatically detects errors by combining static checking, based on theorem proving, and test generation [16].

To the best of our knowledge, CATCHER is the first that combines static exception propagation and search-based testing focusing on the identification of dependency-related bugs, in client programs, caused by misuses of the Java platform's API.

9 CONCLUSIONS

We introduce a verification technique, CATCHER, that combines static exception propagation analysis and search-based testing to effectively and efficiently identify and expose API misuses in client programs. We validate CATCHER against 21 Java applications, targeting misuses of the Java platform's API. Our results show that CATCHER is able to efficiently generate test cases that uncover 243 API misuses leading to crashes. The collected results indicate that CATCHER can reveal more API misuses (77 cases) that would remain undetected by plain EVOSUITE, while also requiring less than 20% of the time spent by EVOSUITE. Overall, static exception propagation analysis and search-based testing combined can significantly improve the detection capability of API misuses, thereby improving the robustness of applications.

In the future, we aim to extend CATCHER along the following dimensions: (i) introduce support for longer exception propagation chains to cover deeply nested API calls, (ii) consider third-party libraries in the analysis to cover the runtime exceptions they introduce, and (iii) extend CATCHER to cover other types of API misuses (e.g., API initialization violations).

ACKNOWLEDGMENTS

This research was partially funded by the EU Horizon 2020 ICT-10-2016-RIA "STAMP" project (No.731529) and the Dutch 4TU project "Big Software on the Run".

REFERENCES

[1] Mithun Acharya and Tao Xie. 2009. Mining API Error-Handling Specifications from Source Code. In *Fundamental Approaches to Software Engineering*, Marsha Chechik and Martin Wirsing (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 370–384.

[2] Nadia Alshahwan, Xinbo Gao, Mark Harman, Yue Jia, Ke Mao, Alexander Mols, Taijin Tei, and Ilya Zorin. 2018. Deploying Search Based Software Engineering with Sapienz at Facebook. In *Search-Based Software Engineering. SSBSE 2018. (LNCS)*, Vol. 11036. Springer, 3–45. https://doi.org/10.1007/978-3-319-99241-9_1

[3] Sven Amann, 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.

[4] Sven Amann, Hoan Anh Nguyen, Sarah Nadi, Tien N Nguyen, and Mira Mezini. 2018. A Systematic Evaluation of Static API-Misuse Detectors. *IEEE Transactions on Software Engineering* (2018). https://doi.org/10.1109/TSE.2018.2827384

[5] Andrea Arcuri. 2018. EvoMaster: Evolutionary Multi-context Automated System Test Generation. In *Software Testing, Verification and Validation (ICST), 2018 IEEE 11th International Conference on.* IEEE, 394–397.

[6] Andrea Arcuri and Lionel Briand. 2014. A Hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability* 24, 3 (2014), 219–250.

[7] Andrea Arcuri and Gordon Fraser. 2013. Parameter tuning or default values? An empirical investigation in search-based software engineering. *Empirical Software Engineering* 18, 3 (2013), 594–623.

[8] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14).* ACM, 259–269. https://doi.org/10.1145/2594291.2594299

[9] Domagoj Babić, Lorenzo Martignoni, Stephen McCamant, and Dawn Song. 2011. Statically-directed Dynamic Automated Test Generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA '11).* ACM, 12–22. https://doi.org/10.1145/2001420.2001423

[10] Arianna Blasi, Alberto Goffi, Konstantin Kuznetsov, Alessandra Gorla, Michael D. Ernst, Mauro Pezzè, and Sergio Delgado Castellanos. 2018. Translating code comments to procedure specifications. In *ISSTA 2018, Proceedings of the 2018 International Symposium on Software Testing and Analysis.* ACM, Amsterdam, Netherlands. https://doi.org/10.1145/3213846.3213872

[11] Martin Bravenboer and Yannis Smaragdakis. 2009. Exception Analysis and Points-to Analysis: Better Together. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis (ISSTA '09).* ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/1572272.1572274

[12] Raymond P.L. Buse and Westley R. Weimer. 2008. Automatic Documentation Inference for Exceptions. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis (ISSTA '08).* ACM, New York, NY, USA, 273–282. https://doi.org/10.1145/1390630.1390664

[13] José Campos, Yan Ge, Nasser Albunian, Gordon Fraser, Marcelo Eler, and Andrea Arcuri. 2018. An empirical evaluation of evolutionary algorithms for unit test suite generation. *Information and Software Technology* 104 (2018), 207–235. https://doi.org/10.1016/j.infsof.2018.08.010

[14] Roberta Coelho, Lucas Almeida, Georgios Gousios, Arie van Deursen, and Christoph Treude. 2017. Exception handling bug hazards in Android. *Empirical Software Engineering* 22, 3 (01 June 2017), 1264–1304. https://doi.org/10.1007/s10664-016-9443-7

[15] Joël Cox, Eric Bouwers, Marko van Eekelen, and Joost Visser. 2015. Measuring dependency freshness in software systems. In *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, Vol. 2. IEEE, 109–118.

[16] C. Csallner and Y. Smaragdakis. 2005. Check 'n' crash: combining static checking and testing. In *Proceedings of the 27th International Conference on Software Engineering, 2005. ICSE 2005.* 422–431. https://doi.org/10.1109/ICSE.2005.1553585

[17] Kalyanmoy Deb, Samir Agrawal, Amrit Pratap, and Tanaka Meyarivan. 2000. A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: NSGA-II. In *International Conference on Parallel Problem Solving From Nature.* Springer, 849–858.

[18] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. 2013. An Empirical Study of Cryptographic Misuse in Android Applications. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security (CCS '13).* ACM, New York, NY, USA, 73–84. https://doi.org/10.1145/2508859.2516693

[19] Lingling Fan, Ting Su, Sen Chen, Guozhu Meng, Yang Liu, Lihua Xu, Geguang Pu, and Zhendong Su. 2018. Large-Scale Analysis of Framework-Specific Exceptions in Android Apps. *CoRR* abs/1801.07009 (2018). arXiv:1801.07009 http://arxiv.org/abs/1801.07009

[20] Gordon Fraser and Andrea Arcuri. 2011. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering.* ACM, 416–419.

[21] Gordon Fraser and Andrea Arcuri. 2013. Whole test suite generation. *IEEE Transactions on Software Engineering* 39, 2 (2013), 276–291.

[22] Gordon Fraser and Andrea Arcuri. 2014. A large-scale evaluation of automated unit test generation using evosuite. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 24, 2 (2014), 8.

[23] Gordon Fraser and Andrea Arcuri. 2015. 1600 faults in 100 projects: automatically finding faults while achieving high coverage with EvoSuite. *Empirical Software Engineering* 20, 3 (2015), 611–639.

[24] Gordon Fraser and Andreas Zeller. 2012. Mutation-driven generation of unit tests and oracles. *IEEE Transactions on Software Engineering* 38, 2 (2012), 278–292.

[25] C. Fu and B.G. Ryder. 2007. Exception-Chain Analysis: Revealing Exception Handling Architecture in Java Server Applications. In *29th International Conference on Software Engineering (ICSE '07).* ACM, 230–239. https://doi.org/10.1109/ICSE.2007.35

[26] I. Garcia and N. Cacho. 2011. eFlowMining: An Exception-Flow Analysis Tool for .NET Applications. In *Fifth Latin-American Symposium on Dependable Computing Workshops (LADCW '11).* IEEE, 1–8. https://doi.org/10.1109/LADCW.2011.18

[27] Gregory Gay. 2017. Generating effective test suites by combining coverage criteria. In *International Symposium on Search Based Software Engineering.* Springer, 65–82.

[28] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why Don't Software Developers Use Static Analysis Tools to Find Bugs?. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13).* IEEE Press, Piscataway, NJ, USA, 672–681.

[29] René Just, Darioush Jalali, and Michael D Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis.* ACM, 437–440.

[30] Maria Kechagia, Marios Fragkoulis, Panos Louridas, and Diomidis Spinellis. 2018. The exception handling riddle: An empirical study on the Android API. *Journal of Systems and Software* 142 (2018), 248 – 270. https://doi.org/10.1016/j.jss.2018.04.034

[31] Maria Kechagia, Dimitris Mitropoulos, and Diomidis Spinellis. 2015. Charting the API minefield using software telemetry data. *Empirical Software Engineering* 20, 6 (01 Dec 2015), 1785–1830. https://doi.org/10.1007/s10664-014-9343-7

[32] Bogdan Korel. 1990. Automated software test data generation. *IEEE Transactions on software engineering* 16, 8 (1990), 870–879.

[33] Christian Lindig. 2016. Mining patterns and violations using concept analysis. In *The Art and Science of Analyzing Software Data.* Elsevier, 17–38.

[34] L. Ma, C. Artho, C. Zhang, H. Sato, J. Gmeiner, and R. Ramler. 2015. GRT: Program-Analysis-Guided Random Testing (T). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE).* IEEE, 212–223. https://doi.org/10.1109/ASE.2015.49

[35] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: multi-objective automated testing for Android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis.* ACM, 94–105.

[36] Phil McMinn. 2004. Search-based software test data generation: a survey. *Software testing, Verification and Reliability* 14, 2 (2004), 105–156.

[37] Rongxin Wu Xuan Xie Shing-Chi Cheung Zhendong Su Ming Wen, Yepang Liu. 2019. Exposing Library API Misuses via Mutation Analysis. In *Proceedings of the 41th International Conference on Software Engineering, 2019. ICSE 2019.*

[38] Parastoo Mohagheghi and Reidar Conradi. 2007. Quality, productivity and economic benefits of software reuse: a review of industrial studies. *Empirical Software Engineering* 12, 5 (01 Oct 2007), 471–516. https://doi.org/10.1007/s10664-007-9040-x

[39] Urko Rueda Molina, Fitsum Meshesha Kifetew, and Annibale Panichella. 2018. Java unit testing tool competition: sixth round. In *The 11th International Workshop on Search-Based Software Testing (SBST).* 22–29. https://doi.org/10.1145/3194718.3194728

[40] Sarah Nadi, Stefan Krüger, Mira Mezini, and Eric Bodden. 2016. Jumping through hoops: why do Java developers struggle with cryptography APIs?. In *Proceedings of the 38th International Conference on Software Engineering.* ACM, 935–946.

[41] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H Pham, Jafar M Al-Kofahi, and Tien N Nguyen. 2009. 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, 383–392.

[42] Juliana Oliveira, Nelio Cacho, Deise Borges, Thaisa Silva, and Fernando Castor. 2016. An Exploratory Study of Exception Handling Behavior in Evolving Android and Java Applications. In *Proceedings of the 30th Brazilian Symposium on Software Engineering (SBES '16).* ACM, New York, NY, USA, 23–32. https://doi.org/10.1145/2973839.2973843

[43] Carlos Pacheco and Michael D. Ernst. 2007. Randoop: Feedback-directed Random Testing for Java. In *Companion to the 22Nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion (OOPSLA '07).* ACM, New York, NY, USA, 815–816. https://doi.org/10.1145/1297846.1297902

[44] Annibale Panichella, Fitsum Kifetew, and Paolo Tonella. 2017. Automated test case generation as a many-objective optimisation problem with dynamic selection

of the targets. *IEEE Transactions on Software Engineering* 99 (2017), 1–37.

[45] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. 2015. Reformulating branch coverage as a many-objective optimization problem. In *Software Testing, Verification and Validation (ICST), 2015 IEEE 8th International Conference on*. IEEE, 1–10.

[46] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. 2018. Incremental Control Dependency Frontier Exploration for Many-Criteria Test Case Generation. In *Search-Based Software Engineering*, Thelma Elita Colanzi and Phil McMinn (Eds.). Springer International Publishing, Cham, 309–324.

[47] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. 2018. A large scale empirical comparison of state-of-the-art search-based test case generators. *Information and Software Technology* 104 (2018), 236–256. https://doi.org/10.1016/j.infsof.2018.08.009

[48] Annibale Panichella and Urko Rueda Molina. 2017. Java Unit Testing Tool Competition - Fifth Round. In *10th IEEE/ACM International Workshop on Search-Based Software Testing (SBST)*. 32–38. https://doi.org/10.1109/SBST.2017.7

[49] Michael Reif, Florian Kübler, Michael Eichberg, and Mira Mezini. 2018. Systematic Evaluation of the Unsoundness of Call Graph Construction Algorithms for Java. In *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops (ISSTA '18)*. ACM, 107–112. https://doi.org/10.1145/3236454.3236503

[50] Martin P. Robillard and Robert Deline. 2011. A Field Study of API Learning Obstacles. *Empirical Softw. Engg.* 16, 6 (Dec. 2011), 703–732. https://doi.org/10.1007/s10664-010-9150-8

[51] Martin P. Robillard and Gail C. Murphy. 2003. Static Analysis to Support the Evolution of Exception Structure in Object-oriented Systems. *ACM Trans. Softw. Eng. Methodol.* 12, 2 (April 2003), 191–221. https://doi.org/10.1145/941566.941569

[52] José Miguel Rojas, José Campos, Mattia Vivanti, Gordon Fraser, and Andrea Arcuri. 2015. Combining multiple coverage criteria in search-based unit test generation. In *International Symposium on Search Based Software Engineering*. Springer, 93–108.

[53] José Miguel Rojas, Mattia Vivanti, Andrea Arcuri, and Gordon Fraser. 2017. A detailed investigation of the effectiveness of whole test suite generation. *Empirical Software Engineering* 22, 2 (2017), 852–893.

[54] Sina Shamshiri, Rene Just, Jose Miguel Rojas, Gordon Fraser, Phil McMinn, and Andrea Arcuri. 2015. Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 201–211.

[55] Sina Shamshiri, José Miguel Rojas, Luca Gazzola, Gordon Fraser, Phil McMinn, Leonardo Mariani, and Andrea Arcuri. 2018. Random or evolutionary search for object-oriented test suite generation? *Software Testing, Verification and Reliability* 28, 4 (2018), e1660. https://doi.org/10.1002/stvr.1660 e1660 stvr.1660.

[56] S. Sinha and M.J. Harrold. 1998. Analysis of programs with exception-handling constructs. In *Proceedings of the International Conference on Software Maintenance*. IEEE, 348–357. https://doi.org/10.1109/ICSM.1998.738526

[57] Diomidis Spinellis and Panagiotis Louridas. 2007. A Framework for the Static Verification of API Calls. *J. Syst. Softw.* 80, 7 (July 2007), 1156–1168. https://doi.org/10.1016/j.jss.2006.09.040

[58] Suresh Thummalapenta and Tao Xie. 2009. Alattin: Mining alternative patterns for detecting neglected conditions. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 283–294.

[59] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 2010. Soot: A Java Bytecode Optimization Framework. In *CASCON First Decade High Impact Papers (CASCON '10)*. IBM Corp., Riverton, NJ, USA, 214–224. https://doi.org/10.1145/1925805.1925818

[60] András Vargha and Harold D Delaney. 2000. A critique and improvement of the CL common language effect size statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics* 25, 2 (2000), 101–132.

[61] 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.

[62] Joachim Wegener, André Baresel, and Harmen Sthamer. 2001. Evolutionary test environment for automatic structural testing. *Information and Software Technology* 43, 14 (2001), 841–854.

[63] Sai Zhang, David Saff, Yingyi Bu, and Michael D. Ernst. 2011. Combined Static and Dynamic Automated Test Generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA '11)*. ACM, 353–363. https://doi.org/10.1145/2001420.2001463

[64] Y. Zhou, R. Gu, T. Chen, Z. Huang, S. Panichella, and H. Gall. 2017. Analyzing APIs Documentation and Code to Detect Directive Defects. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. ACM, 27–37. https://doi.org/10.1109/ICSE.2017.11