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


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 Documentation is not adequate, as it is usually either outdated [10], defective [64], or just ignored by the developers of client applications [50].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.\(^1\)

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 sanitation) 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 1: JDK API misuse in joda-time, issue #319](https://github.com/mkechagia/Catcher)
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 `IOException`, 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]:

$$bc_{m_i} = al(t_j, b_k) + \text{norm}(bd(t_j, b_k))$$

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 [32], such as triggering the propagated exceptions.

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

$$f(m_1) = \min(bc_{m_1}), \max(ic_{m_1}), \max(oc_{m_1})$$

$$\ldots$$

$$f(m_n) = \min(bc_{m_n}), \max(ic_{m_n}), \max(oc_{m_n})$$

### 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
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 verify the usages of other third-party APIs, exposing the misuses. To the best of our knowledge, we are the first to apply coverage-based criteria (e.g., branch coverage). However, covering unit-test generation tools (such as EvoSuite) of the client applications and they can be covered by traditional generators regarding their capability to generate test cases able 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. 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 contest [39, 46], 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 et 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,

### Table 1: Java projects in our Benchmark

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Version</th>
<th>Files</th>
<th>LOC</th>
<th># Candi. Misuses</th>
</tr>
</thead>
<tbody>
<tr>
<td>BCCL</td>
<td>bcrest</td>
<td>6.2</td>
<td>488</td>
<td>39K</td>
<td>223</td>
</tr>
<tr>
<td>CLI</td>
<td>commons-cli</td>
<td>1.4</td>
<td>50</td>
<td>7K</td>
<td>114</td>
</tr>
<tr>
<td>CODEC</td>
<td>commons-codec</td>
<td>1.12</td>
<td>124</td>
<td>20K</td>
<td>580</td>
</tr>
<tr>
<td>COLL</td>
<td>commons-collections</td>
<td>4.2</td>
<td>535</td>
<td>63K</td>
<td>213</td>
</tr>
<tr>
<td>COMP</td>
<td>commons-compress</td>
<td>1.17</td>
<td>352</td>
<td>43K</td>
<td>587</td>
</tr>
<tr>
<td>LANG</td>
<td>commons-lang</td>
<td>3.7</td>
<td>323</td>
<td>76K</td>
<td>1024</td>
</tr>
<tr>
<td>MATH</td>
<td>commons-math</td>
<td>3.6</td>
<td>1617</td>
<td>209K</td>
<td>411</td>
</tr>
<tr>
<td>EASY</td>
<td>easymock</td>
<td>3.6</td>
<td>204</td>
<td>14K</td>
<td>217</td>
</tr>
<tr>
<td>GSON</td>
<td>gson</td>
<td>2.85</td>
<td>206</td>
<td>23K</td>
<td>157</td>
</tr>
<tr>
<td>HAMC</td>
<td>hamcrest-core</td>
<td>1.3</td>
<td>152</td>
<td>7K</td>
<td>34</td>
</tr>
<tr>
<td>JACK</td>
<td>jackson-datalind</td>
<td>2.96</td>
<td>919</td>
<td>114K</td>
<td>251</td>
</tr>
<tr>
<td>JAVS</td>
<td>javassist</td>
<td>3.23.1</td>
<td>527</td>
<td>82K</td>
<td>313</td>
</tr>
<tr>
<td>JCOM</td>
<td>jcommander</td>
<td>1.71</td>
<td>139</td>
<td>6K</td>
<td>68</td>
</tr>
<tr>
<td>JFCH</td>
<td>jfrechard</td>
<td>1.50</td>
<td>990</td>
<td>134K</td>
<td>681</td>
</tr>
<tr>
<td>JODA</td>
<td>joda-time</td>
<td>2.10</td>
<td>330</td>
<td>16K</td>
<td>1053</td>
</tr>
<tr>
<td>JOPT</td>
<td>jopt-sample</td>
<td>5.04</td>
<td>192</td>
<td>9K</td>
<td>107</td>
</tr>
<tr>
<td>NATT</td>
<td>natty</td>
<td>0.13</td>
<td>27</td>
<td>3K</td>
<td>33</td>
</tr>
<tr>
<td>NEO4</td>
<td>neo4j-java-driver</td>
<td>1.62</td>
<td>510</td>
<td>52K</td>
<td>784</td>
</tr>
<tr>
<td>SHIRO</td>
<td>shiro-core</td>
<td>1.32</td>
<td>653</td>
<td>31K</td>
<td>223</td>
</tr>
<tr>
<td>XJB</td>
<td>xwkle-commons-job</td>
<td>10.6</td>
<td>67</td>
<td>3K</td>
<td>616</td>
</tr>
<tr>
<td>XTEX</td>
<td>xwkle-commons-text</td>
<td>10.6</td>
<td>3</td>
<td>101</td>
<td>11</td>
</tr>
</tbody>
</table>

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. 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.
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) \( \times \) 25 (repetitions) \( \approx \) 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 \( \approx \) 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 CPU 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 answer 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 \( A_{12} \) 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).
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 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, 55], 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 (xwik1-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 EvoSuite 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 the WeakMutator. 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 netty. 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 (Δ12 > 0.90).

### Results of RQ3

(whose procedure is based in Section 4.4), we identified
Table 4: Overlap between Catcher and EvoSuite regarding the unique crash-related API misuses detected across 25 repetitions.

<table>
<thead>
<tr>
<th>Project</th>
<th>Catcher</th>
<th>Catcher</th>
<th>EvoSuite</th>
<th>EvoSuite</th>
<th>EvoSuite</th>
<th>Catcher</th>
</tr>
</thead>
<tbody>
<tr>
<td>BCEL</td>
<td>7</td>
<td>2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>CODEC</td>
<td>9</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>COLL</td>
<td>9</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>COMP</td>
<td>16</td>
<td>18</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>LANG</td>
<td>23</td>
<td>-</td>
<td>9</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>MATH</td>
<td>8</td>
<td>4</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>EASY</td>
<td>11</td>
<td>5</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>GSON</td>
<td>8</td>
<td>8</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>JACK</td>
<td>5</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>JAVS</td>
<td>2</td>
<td>2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>JCOM</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>JFCH</td>
<td>23</td>
<td>-</td>
<td>4</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>JODA</td>
<td>11</td>
<td>14</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>JOPT</td>
<td>3</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>NATT</td>
<td>4</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>NEO4</td>
<td>8</td>
<td>9</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SHIRO</td>
<td>6</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>XJOB</td>
<td>9</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Total</td>
<td>163</td>
<td>80</td>
<td>2</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

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

(a) Class KthSelector from apache-commons-math

Java.lang.ArrayIndexOutOfBoundsException at java.util.Arrays.sort(Arrays.java:440)
at java.util.Arrays.rangeCheck(Arrays.java:120)
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];
    intArray0 = new int[3];
    KthSelector kthSelector0 = new KthSelector();
    int[] intArray8 = new int[] {11, intArray8[1] = 11, intArray8[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. 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.

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
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 $\bar{A}_{12}$ 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. 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.

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 inter-procedural 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 [29, 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”.

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


