Test Mimicry to Assess the Exploitability of Library Vulnerabilities

Hong Jin Kang
Singapore Management University
Singapore
hjkang.2018@phdcs.smu.edu.sg

Truong Giang Nguyen
Singapore Management University
Singapore
gtnguyen@smu.edu.sg

Bach Le
University of Melbourne
Australia
bach.le@unimelb.edu.au

Corina S. Păsăreanu
Carnegie Mellon University and
NASA Ames Research Center
USA
pcorina@cmu.edu

David Lo
Singapore Management University
Singapore
davidlo@smu.edu.sg

ABSTRACT

Modern software engineering projects often depend on open-source software libraries, rendering them vulnerable to potential security issues in these libraries. Developers of client projects have to stay alert of security threats in the software dependencies. While there are existing tools that allow developers to assess if a library vulnerability is reachable from a project, they face limitations. Call graph-only approaches may produce false alarms as the client project may not use the vulnerable code in a way that triggers the vulnerability, while test generation-based approaches faces difficulties in overcoming the intrinsic complexity of exploiting a vulnerability, where extensive domain knowledge may be required to produce a vulnerability-triggering input.

In this work, we propose a new framework named *Test Mimicry*, that constructs a test case for a client project that exploits a vulnerability in its library dependencies. Given a test case in a software library that reveals a vulnerability, our approach captures the program state associated with the vulnerability. Then, it guides test generation to construct a test case for the client program to invoke the library such that it reaches the same program state as the library's test case. Our framework is implemented in a tool, Transfer, which uses search-based test generation. Based on the library's test case, we produce search goals that represent the program state triggering the vulnerability. Our empirical evaluation on 22 real library vulnerabilities and 64 client programs shows that Transfer outperforms an existing approach, Siege; Transfer generates 4x more test cases that demonstrate the exploitability of vulnerabilities from client projects than Siege.

CCS CONCEPTS

• **Software and its engineering** → **Software libraries and repositories**; **Search-based software engineering**.

KEYWORDS

Library Vulnerabilities, Search-based Test Generation

ACM Reference Format:
Hong Jin Kang, Truong Giang Nguyen, Bach Le, Corina S. Păsăreanu, and David Lo. 2022. Test Mimicry to Assess the Exploitability of Library Vulnerabilities. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '22), July 18–22, 2022, Virtual, South Korea.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3533767.3534398

1 INTRODUCTION

Software engineering projects often depend on open-source software libraries [46, 55, 65]. As vulnerabilities in a project's library dependencies may be exploited by attackers of the project, developers have to understand their project's dependencies and update them whenever library vulnerabilities are discovered. For example, the recent Log4Shell vulnerability, which affected millions of devices, required client developers to update their applications to use the latest version of the Log4J library quickly [4, 11]. Moreover, as the clients of a library include other libraries, a vulnerability in one library would transitively propagate throughout the ecosystem of libraries and their clients.

After library vulnerabilities have been fixed and publicly disclosed, client developers are advised to update their dependencies to use the new, non-vulnerable versions of libraries. However, studies have shown that client developers are reluctant to update their dependencies. Many alerts regarding vulnerable dependencies are false alarms [55], and developers may be wary of breaking changes from library updates. This leaves client projects open to exploitation of vulnerabilities in library dependencies [28, 46, 51, 65].

To address this problem, there have been proposed techniques that assess the *reachability* of the vulnerable code (e.g. function) from the client project, allowing developers and security researchers to better assess a vulnerability's exploitability from the client project. Existing techniques use call graph analysis to determine if the vulnerable code is called from the client project [32, 40, 58]. As control-flow within functions is not considered, existing techniques produce false alarms [32]. Fundamentally, these tools are limited since they check only if a vulnerable function may be *called*, but do not determine if the client projects are able to construct the inputs that trigger the vulnerability [32, 40, 58].

Recently, Iannone et al. [40] proposed SIEGE, a tool that automatically generates test cases demonstrating the exploitability of library vulnerabilities for client projects [40]. Given a description of the vulnerability, which is manually determined from a vulnerability-fixing commit, SIEGE targets the coverage of a single line of library code indicated. SIEGE generates test cases of the client projects that transitively execute the line of code, thus, providing evidence that the library vulnerability can be reached from the client project. While SIEGE could confirm the exploitability of some vulnerabilities, it is limited by its inability to overcome the intrinsic complexity of exploiting vulnerabilities; specific domain knowledge is required for triggering many vulnerabilities [40].

Recreating the triggering conditions of a vulnerability may be challenging because of the domain knowledge required. Take CVE-2019-12402 [3] in Apache Tika as an example: an attacker can trigger a denial-of-service attack by providing a zip file quine[1], a specially-crafted zip file that is unzipped to produce itself as output. It is extremely hard to build a zip file quine with random mutations (as in fuzzing) even with some guidance.

To overcome the challenging requirement of extensive domain knowledge, we propose to leverage test cases of the vulnerable, open-source libraries, particularly the test cases that accompany the vulnerability fixes. Specifically, we propose a new framework, which we term *Test Mimicry*, depicted in Figure 1. Expert domain knowledge and the conditions (e.g., specific inputs, program state) of triggering a vulnerability are captured in the test cases written by the domain experts. Rather than blindly generating test cases, we generate test cases of the client project that invokes the vulnerable method with the same arguments as the library's vulnerability-witnessing test cases. Rather than designing oracles to detect if a vulnerability has been triggered, we detect the replication of the program state reached in the vulnerability-witnessing test case.

Concretely, our framework uses the vulnerability-witnessing test case from the library's code, denoted as $\mathcal{L}^{\mathcal{T}}$, to generate a test case, denoted as $C^{\mathcal{T}}$, for code in a client project that *mimics* $\mathcal{L}^{\mathcal{T}}$. $C^{\mathcal{T}}$ should test the client program such that the library exhibits the same behavior when it was tested with $\mathcal{L}^{\mathcal{T}}$. If the same program state reached by $\mathcal{L}^{\mathcal{T}}$ can be reproduced, then the vulnerability can be triggered from the client program. If so, then we have evidence that the vulnerability is exploitable from the client project.

To this end, we implement a tool, TRANSFER. TRANSFER targets client projects of open-source Java libraries. Given a library function, *VF*, associated with a known vulnerability, TRANSFER executes the library test case, $\mathcal{L}^{\mathcal{T}}$, that demonstrates how the vulnerability can be triggered. After identifying the program state relevant to the vulnerability, TRANSFER extracts the *triggering conditions*, σ, which are satisfied by reaching the same program state from a generated test case. TRANSFER uses an evolutionary algorithm to generate test cases, directing it to transitively invoke *VF* through the client program and favors test cases closer to satisfying σ. Finally, TRANSFER outputs a test case if it is sufficiently close to satisfying σ.

We evaluate TRANSFER by analyzing 22 real library vulnerabilities from a wide range of domains (e.g. JSON parsing, file compression) and types of vulnerabilities (e.g. XXE injections, unhandled exceptions). Analyzing 64 real client programs obtained from GitHub, we find that a library vulnerability can be exploited from 42 of them. While SIEGE produces exploits for 5 client programs, TRANSFER generates exploits for 23 client programs.

[1]https://www.cvedetails.com/cve/CVE-2019-10094/

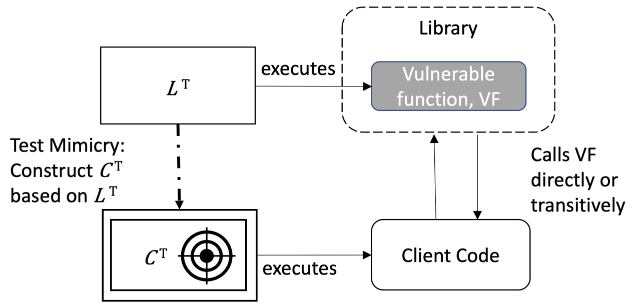

Library
executes
Vulnerable function, VF
L^{T}
Test Mimicry: Construct C^{T} based on L^{T}
Calls VF directly or transitively
C^{T}
executes
Client Code

Figure 1: Test mimicry: Constructing a test case, $C^{\mathcal{T}}$, for the library's client that demonstrates the same library vulnerability as witnessed by a library's test case, $\mathcal{L}^{\mathcal{T}}$.

In this study, we make the following contributions:

- We propose the novel framework of **test mimicry**, addressing the problem of missing domain knowledge when assessing the exploitability of library vulnerabilities.
- We implement **TRANSFER**, which guides test generation to reproduce the same program state as the library's test case when invoking the vulnerable function.
- Our **evaluation** shows TRANSFER successfully generates 4x more exploits than the state-of-the-art approach SIEGE [40].

2 BACKGROUND AND MOTIVATION

Our work builds on prior studies on software composition analysis and search-based test generation. In this section, we describe them.

2.1 Software Composition Analysis

Software composition analysis is a domain related to the identification and replacement of vulnerable dependencies of software projects [32, 58]. Many solutions enumerate through a project's dependencies to detect potentially vulnerable libraries (i.e., looking up the versions of the project's dependencies against a database of known vulnerable library versions). From the source code of the project, a call graph is constructed to determine if a vulnerable library function is reachable. These analyses produce false positives as static call graphs may contain calls that do not occur at runtime [35, 61]. Hence, existing approaches [32, 58] complement the static analysis with dynamic analysis by running the client projects' test cases to construct call graphs, which reduces the number of false positives. These techniques are limited by the test coverage of the client projects, which may be low [43, 44]. Fundamentally, call graph-based approaches are limited as they do not consider control-flow or check that the inputs for triggering the vulnerability can be passed from client programs [32, 40, 58].

SIEGE [40] is a search-based test generator that produces exploits on client programs that executes vulnerable library code. These exploits are in the form of test cases that reveal how the library vulnerabilities can be exploited from the client programs. Exploits

Algorithm 1: Simplified version of a genetic algorithm for generating test cases

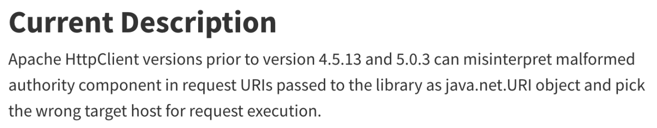
```
Inputs: 1. goals, the search goals
        2. M, the population size
        3. search_budget, amount of time to run
Output: tests, test cases
P_0 = construct_random_population(M)
k = 0
fitness = compute_fitness(pop, goals)
best_fitness = max(fitness)
while best_fitness < 1 and time spent < search_budget do
    k = k + 1
    // offsprings through mutations and crossover
    offsprings = generate_offspring(P_{k-1})
    // evaluate fitness and pick top M tests
    P_k = P_{k-1} ∪ offsprings
    fitness = compute_fitness(P_k, goals)
    P_k = select_top_M(P_k, fitness)
end
return P_k
```

produced by Siege and our tool, Transfer, act as evidence of the exploitability of the library vulnerability from the client projects.

2.2 Search-Based Test Generation

Search-based techniques have been proposed for generating test cases to satisfy a specified search criterion. EvoSuite generates test cases that achieve high coverage for Java programs [33]. Siege [40] assesses the exploitability of library vulnerabilities from client programs generating a test case for the client program, guided by the criterion of executing a vulnerable line of library code given as an input vulnerability description. Algorithm 1 shows a simplified version of a genetic algorithm for generating test cases. Starting with a population of randomly generated test cases, a fitness value for each test case is computed with respect to the search goals (e.g., total code coverage for EvoSuite, how close the test is to covering the vulnerable line of code for Siege). While the search budget has not been exhausted and the optimal fitness value has not been reached, the genetic algorithm produces a new generation of test cases through mutations and crossovers on the previous generation of the test cases. The top M test cases are selected based on their fitness values to populate the next generation. As such, test cases with poor fitness are removed from the population.

A challenge faced by search-based test generators is the reproduction of complex behaviors. It is difficult for unguided, random test case generation to produce test cases that invoke complex behaviors. To generate test cases that invoke more complex behaviors, many techniques have been proposed. One such method is seeding [60], which uses existing knowledge about the program to help solve the search process. For example, string and numerical literals within the program are extracted into a constants pool. Seeding increases the likelihood of the test generator using these values, allowing it to pass difficult conditional checks. A related technique is *test carving* [31, 60]. Test carving was proposed to convert larger


Current Description

Apache HttpClient versions prior to version 4.5.13 and 5.0.3 can misinterpret malformed authority component in request URIs passed to the library as java.net.URI object and pick the wrong target host for request execution.

Figure 2: The information of a vulnerability given in NVD. The high-level description of the vulnerability makes assessing its possible impact difficult.

system tests to a set of smaller unit tests for the same project. The technique extracts parts of the program state reached in the system tests as they are executed, capturing potentially reusable objects that can be recreated when constructing new test cases. The object states that comprise the program state may be difficult to recreate, and carving allows the construction of new test cases that starts from the same program state.

In this work, we build Transfer using the infrastructure and tooling provided by EvoSuite. Due to its maturity [12, 30], Transfer uses EvoSuite's implementation of the genetic algorithm, including the crossover and mutation operators. Unlike EvoSuite, Transfer seeks to reproduce, from client programs, the behavior demonstrated by the vulnerability-witnessing test case. Transfer's search criteria are dynamically determined from execution of the vulnerability-witnessing test case. Transfer targets the same vulnerable library function executed by the test case, producing new test cases that satisfies the triggering conditions extracted from its carved program state.

2.3 Motivating Example

From an alert about a new vulnerability in a library, e.g. CVE-2020-13956, a developer of a project (which may itself be another library) using the vulnerable library (e.g. Apache HttpComponents) is unsure if the vulnerability can be exploited. Figure 2 shows the vulnerability's high-level description, indicating that the vulnerable behavior occurs when a "misinterpreted authority component in request URIs" is input to the library. Without better understanding the vulnerability's exploitability and knowing that many alerts about vulnerable dependencies are false alarms [56], the developer may not prioritize the library update, leaving the library vulnerability in the project open to exploitation.

On the other hand, **with the test case generated by Transfer**, the developer has evidence of the exploitability of the vulnerability from the client project. As seen in Figure 3, the test case shows the client project class (`HttpClient`) and function (`execute`) that transitively calls the vulnerable function, how the function from the client program may be invoked to trigger the vulnerability (i.e. the construction of multiple classes from the client project and a concrete example of a malformed URL triggering the vulnerability, *http://blah@goggle.com:80@google.com*). While Siege struggles to construct a malformed URL, Transfer uses the example of the malformed URL from the library's test case. Providing evidence of the exploitability of a vulnerable may motivate developers to update their library dependencies.

Given the growing prevalence of library vulnerabilities, their widespread impact, and the importance of detecting them, reducing

```
Config config = new Config();
BasicClassicHttpRequest httpRequest =
  new BasicClassicHttpRequest("/", null,
  "http://blah@goggle.com:80@google.com/");
HttpClient httpClient = new HttpClient(config);
try {
  httpClient.execute(httpRequest);
  fail("Expecting IOException");
} catch(IOException e) {
  verifyException("CloseableHttpClient", e);
}
```

Figure 3: Snippet of a test case generated by Transfer, with changes made for conciseness. The test case demonstrates how the client program can transitively invoke *VF* with inputs that triggers the vulnerability.

the difficulty of generating exploits for even a proportion of vulnerabilities would already be helpful. Moreover, writing test cases for bug fixes is considered a good software development practice [16]. Still, we assess the feasibility of using test cases from libraries. We looked for vulnerabilities reported in vulnerability databases [9, 10]. From the entries between March 2017 and March 2021, we sampled 780 entries containing a reference to a GitHub commit fixing the vulnerability. From the 780 entries, we identified 233 vulnerabilities from libraries. Of the 233 vulnerabilities, 121 (over 51%) of the commits include a test case, showing that a majority of vulnerabilities are fixed with test cases. This indicates that test mimicry is likely to work for a large number of vulnerabilities.

3 TEST MIMICRY

In this section, we describe the details of our tool, Transfer.

3.1 Objectives and Problem Formulation

From a known vulnerability in a library and a function associated with the vulnerability, the *vulnerable function*, *VF*, a client program is a program from a different project using the library. Given a test case, $\mathcal{L}^{\mathcal{T}}$, that witnesses the vulnerability, our goal is to assess if the library vulnerability can be exploited in the client program by deriving a test case $C^{\mathcal{T}}$ for the client program that mimics $\mathcal{L}^{\mathcal{T}}$. We refer to $C^{\mathcal{T}}$ as the *exploit* (from the client program) and $\mathcal{L}^{\mathcal{T}}$ as the *vulnerability-witnessing test case* (from the library code). We consider that a library vulnerability can be **exploited** from the client project if $C^{\mathcal{T}}$ can be generated. $C^{\mathcal{T}}$ is a test case invoking public functions of the client program and transitively executes *VF* to exhibit the same behavior witnessed by $\mathcal{L}^{\mathcal{T}}$.

3.2 Approach

Figure 4 presents an overview of our tool, Transfer. Transfer takes as input 1) the vulnerability-witnessing test case, $\mathcal{L}^{\mathcal{T}}$, from an open-source library, 2) the vulnerable library function, *VF*, and 3) the code of the client project that uses the vulnerable library. After instrumenting and executing $\mathcal{L}^{\mathcal{T}}$, Transfer extracts the *triggering conditions*, σ, from the carved program state. Transfer generates a test case (that executes the client program) that satisfies σ if it finds one within the search budget.

3.2.1 *Execution of $\mathcal{L}^{\mathcal{T}}$.*

Test mimicry relies on information from $\mathcal{L}^{\mathcal{T}}$. Transfer instruments the library code to carve $\mathcal{L}^{\mathcal{T}}$, Transfer executes $\mathcal{L}^{\mathcal{T}}$ to invoke the vulnerable library function, *VF*, and collects the sequences of function calls and arguments to construct and initialize objects. Based on inputs and outputs of *VF*, the relevant parts of the program state are extracted ($\textcircled{1}$ in Figure 4).

3.2.2 *Search Goals.*

To achieve the high-level objective of reproducing the vulnerable behavior, Transfer attempts to direct test case generation towards code that executes *VF* and bring the program to the same state as $\mathcal{L}^{\mathcal{T}}$. To do so, Transfer constructs the search goals ($\textcircled{2}$ in Figure 4) that will guide the test generator. Transfer uses two types of search goals: line coverage goals and a goal that represents the vulnerability's triggering conditions. After computing a static call graph, Transfer determines the paths from the client program that lead to *VF*. For the functions on the paths, line coverage goals are constructed for the lines in the functions. From the carved object states obtained from the execution of $\mathcal{L}^{\mathcal{T}}$, Transfer extracts the vulnerability's *triggering conditions*, σ.

Each search goal type is associated with a fitness function that estimates the distance of a test case from satisfying the goal. For a line coverage goal, the fitness functions measure how close the test case came to covering the line. For the triggering conditions, the fitness function is an estimate of similarity of the program state when invoking VF to σ (described in Section 3.3).

Line coverage goals for directed test generation. While our objective is to satisfy the conditions of triggering the vulnerability when transitively invoking *VF*, generating a test case that invokes *VF* through the client program can be challenging. To address this challenge, Transfer constructs line coverage goals for the lines of the functions in the call graph between the client program and *VF*, directing the search towards test cases that are closer (covering code on paths in the call graph that end at *VF*) to invoking *VF*.

Transfer uses a standard fitness function for line coverage using branch distance [13, 50]. Based on the control dependencies of the line, the branch distance heuristically estimates how close a test case is to taking a correct branch that the line depends on (e.g. how close the branch predicate is to becoming true) when an undesired control-flow path is taken [13, 50].

Vulnerability-triggering conditions. Transfer carves $\mathcal{L}^{\mathcal{T}}$ to obtain the program state associated with the triggering of the vulnerability, from which it extracts values and properties of the objects to form the vulnerability triggering conditions. Test cases are generated and evolved towards satisfying the triggering conditions.

To guide test generation towards the vulnerable behavior, the fitness functions favor test cases that, through functions of the client project, invokes *VF* with inputs and output having states that come closer to satisfying σ. We provide more details in Section 3.3.

3.2.3 *Evolutionary Test Generation.*

For test generation, Transfer uses an evolutionary algorithm to favor the generation of fitter test cases ($\textcircled{3}$ in Figure 4, with a simplified version shown in Algorithm1). Transfer seeds the object pool with the carved objects from $\mathcal{L}^{\mathcal{T}}$, allowing Transfer to invoke the function calls to

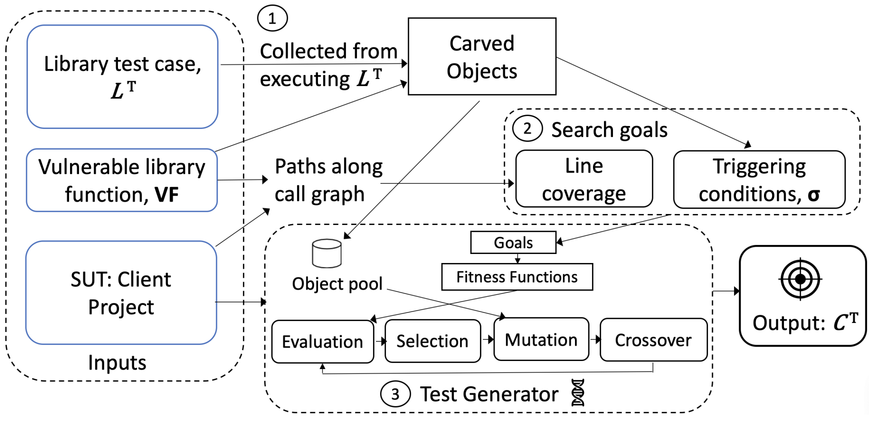


Figure 4: Overview of Transfer. From the vulnerability-witnessing test case from the library, Transfer produces a exploit for the client project – the Software Under Test (SUT).

create complex objects from the library. Due to EvoSuite's maturity and effectiveness [12, 30], we use its representation (i.e., a sequence of function calls) and implementation of test chromosomes, crossover, and mutation operators in Transfer. Transfer uses the multi-objective DYNAMOSA [54] algorithm, shown to be the state-of-the-art for search-based test generation.

The DYNAMOSA [54] algorithm was proposed for test generators to fully achieve the coverage of each individual goal. Other algorithms may instead converge in a local optimum that minimizes its distance from multiple objectives, but does not necessarily satisfy any individual goal [54]. This characteristic of DYNAMOSA makes it suitable for our work as our primary focus is to reproduce the program state of the vulnerability regardless of how close the test case is to covering the other (line coverage) goals.

Transfer's effectiveness stems from *test mimicry*, i.e. dynamically constructing search goals determined from the execution of $\mathcal{L}^{\mathcal{T}}$. In Transfer, the test case with the highest fitness with respect to σ is produced as the output, $C^{\mathcal{T}}$, if its fitness is sufficiently high (set to a fitness threshold of 0.9 in our experiments).

Once the triggering conditions are met or the search budget has been used up, Transfer outputs the test case by considering only the search goal representing the triggering conditions. We do not consider line coverage goals when determining the best test case, as an exploit, $C^{\mathcal{T}}$, may not have covered all lines in the functions between the client program and *VF* in the call graph.

3.3 Satisfying a Vulnerability's Triggering Conditions

Transfer's primary objective is the reproduction of the behavior revealed by the $\mathcal{L}^{\mathcal{T}}$. To detect that a test case has reproduced the same behavior, Transfer uses the program state relevant to the input and output of *VF* carved from the execution of $\mathcal{L}^{\mathcal{T}}$, extracting a set of conditions as the *triggering conditions*, σ. σ abstracts over the program state reached by $\mathcal{L}^{\mathcal{T}}$.

To detect that a generated test case satisfies σ, Transfer compares the inputs and outputs of *VF* with σ. If the test case successfully (transitively through the client program) invokes *VF* with inputs and outputs that match σ, then σ is satisfied and we consider that the test case has invoked *VF* with the same behavior as $\mathcal{L}^{\mathcal{T}}$. We emphasize that the test generator **does not produce test cases that directly call** *VF*. Instead, Transfer produces test cases that execute the client program, i.e., *VF* is transitively invoked through the client program.

For example, based on the vulnerability discussed in Section 2.3, Transfer identifies that the vulnerability is triggered from the program state reached by passing "blah@goggle.com:80@google.com" as an input to a function, `URIUtils.extractHost` used by the library to extract the host of a URL. Transfer generates test cases of the client program that calls the library functions that may directly or transitively invoke the `URIUtils.extractHost` function.

During test generation on the client project, we compute the fitness of the test case with respect to the triggering conditions. The fitness function estimates the similarity of the program state reached by the generated test case as compared to $\mathcal{L}^{\mathcal{T}}$.

Vulnerability triggering conditions. The triggering conditions, σ, are predicates over the values of the input and output of the *VF*. To detect if σ is satisfied, we can view it as the comparison of the relevant objects captured in σ during the execution of $\mathcal{L}^{\mathcal{T}}$ ((2) in Figure 4) and their corresponding values currently observed during the invocation of *VF* (as part of test generation, (3) in Figure 4). The input of *VF* is its method receiver and arguments, and its output is either its return value or an exception thrown during its execution. To simplify our description, let us define ***actuals*** to refer to the inputs and output of *VF* when transitively invoked (through the client program) by a generated test case. We define ***references*** as the corresponding values in σ, which capture the input and output values observed during the execution of $\mathcal{L}^{\mathcal{T}}$. During test generation, if all of *actuals* match their corresponding parts in *references*, then σ is satisfied and $C^{\mathcal{T}}$ has been found.

The similarity between *actuals* and *references* is the average similarity between the individual elements of *actuals* and their corresponding element in *references*. The best similarity value is 1 and the worst similarity value is 0.

$$similarity(actuals, references) = \frac{1}{N} \times \sum_{i=1}^{N} similarity(actuals[i], references[i])$$

Algorithm 2: *similarity_stringlike(actual, reference)*: Computing the similarity of *actual* and *reference* when they are string-like (e.g. `String`, `char[]`)

```
Inputs: 1. actual, one of the objects in actuals.
        2. reference, one of the objects in references.
Output: similarity between actual and reference, a float
        between 0 and 1.
function similarity_stringlike(actual, reference):
1 edit_dist = edit_distance(actual, reference)
2 max_len = max(length(actual), length(reference))
3 return max_len > 0 ? 1 - edit_dist / max_len : 0
```

For comparison between individual elements of *actuals* and *references*, Transfer compares them based on their type. We consider the following types of values:

- Primitive and enumeration values: For primitive values (e.g., `long`, `int`, `char`) as well as values of enums, Transfer directly compares their values.
- String-like values: For values of types `byte[]`, `char[]`, `String`, Transfer compares them by computing the edit distance normalized by the maximum length of the values.
- Objects: To compare two objects, Transfer compares their externally observable states [27].
- Files: For files, Transfer treats them as objects, however, any files read by $\mathcal{L}^{\mathcal{T}}$ are also made available for reading to $C^{\mathcal{T}}$. File names are stored in the constant pool to allow test cases to have a greater chance of reading the file.
- Exceptions: Exceptions can be thrown by *VF*. Transfer considers any exception as the output of *VF*, comparing its type and exception message.

3.3.1 *Comparing Primitive or Enumeration Values.*

Transfer directly compares primitive values and enums. Primitive types include long, int, short, double, float, char, boolean. If *actual* matches *reference*, then a similarity value of 1 is returned, otherwise a similarity value of 0 is returned.

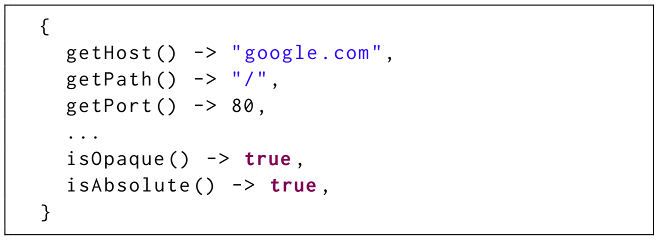
3.3.2 *Comparing String-like Values.*

For values that are string-like (i.e., `char[]`, `byte[]`, `String`), As shown in Algorithm 2, during the generation of test cases, Transfer computes the edit distance between the two string-like values. The fewer edits needed to convert one value to the other, the more similar they are. The similarity is then obtained by subtracting the edit distance normalized by the length of the longer value from 1.

3.3.3 *Comparing Objects.*

For objects, Transfer characterizes them by their externally observable state, using inspector functions [27] defined on the class. Inspector functions refer to public methods that return a value and are side-effect free (determined through a simple static analysis [33]). The state of each object is then constructed using the values returned by the inspectors. An example object state for a `java.net.URI` object is given in Figure 5.

Similarity of two objects Algorithm 3 shows the computation of the similarity between *actual* and *reference*. First, Transfer checks for `null` for both the *actual* and *reference* objects (Lines 1-9).



```
{
  getHost() -> "google.com",
  getPath() -> "/",
  getPort() -> 80,
  ...
  isOpaque() -> true,
  isAbsolute() -> true,
}
```

Figure 5: Simplified example of the object state of a URI object, identified through a mapping of its inspector functions to their return values

Algorithm 3: *similarity(actual, reference)*: computing the similarity of two objects, *actual* and *reference*.

```
Inputs: 1. actual, one of the objects in actuals.
        2. reference, one of the objects in references.
Output: similarity between actual and reference, a float
        between 0 and 1.
function similarity(actual, reference):
1  if reference == null and actual == null then
2      return 1
3  end
4  if reference == null or actual == null then
5      return 0
6  end
7  if actual.getClass() ≠ reference.getClass() then
8      return 0
9  end
10 ins = get_inspectors(actual)
11 if ins is not empty then
12     obj_sim ← $\sum_{in \in ins}$ similarity(actual, reference, in)
13     obj_sim ← $\frac{1}{|ins|} \times$ obj_sim
14     return obj_sim
15 else
16     return 0
17 end
```

If both of them are `null`, then Transfer considers them to be a match, otherwise, if either *actual* or *reference* is `null`, Transfer assigns the lowest similarity score (similarity=0).

Transfer compares inspectors only if both of them are non-null and are objects of the same class (Lines 10 to 17). The similarity of the objects is computed as the average similarity between the inspectors of the objects, which will be described in the next paragraph. *similarity(actual, reference)* returns a value between 0 to 1, where 1 is the highest similarity. Transfer conservatively rejects the match if no inspectors are found (Line 16).

Similarity of two objects w.r.t an inspector. Algorithm 4 shows the computation of the similarity of *actual* and *reference* with respect to a single, given inspector. Its return value is between 0 and 1, inclusively. For each inspector function, Transfer uses a different similarity function based on the function return type. If the return type of the inspector is primitive, then the similarity of the two values is computed using *similarity_primitive* (described in Section 3.3.1). If the return type is string-like, then *similarity_stringlike* (Section 3.3.2) is invoked on the return values of the inspectors instead. Otherwise, if the inspector returns another object, we recursively invoke *similarity* (Figure 3). To avoid infinite recursion (e.g. a function returning the another object of the same class), a depth parameter can be configured in Transfer. For example, the `normalize()` function of a `java.net.URI` returns another `java.net.URI` object. By default, we limit our consideration of objects only up to a depth of two.

3.3.4 Comparing Files. For files that are accessed during the execution of the vulnerability-witnessing test, $\mathcal{L}^{\mathcal{T}}$, they are treated as objects (e.g. typically files are accessed through an `InputStream` object or a `File` object, and can be treated as such). The files on the filesystem are copied and made available during the generation of $C^{\mathcal{T}}$. The filenames are added as strings to the constants pool, which allows Transfer to use the filename when generating test cases. As they are treated as objects, Transfer uses the same similarity function (Algorithm 3).

3.3.5 Comparing Exceptions. Transfer compares exceptions thrown by *VF* using their types (e.g. `NullPointerException`) and exception messages. While other studies [29] have used more sophisticated methods of comparing exceptions, we find that the use of the exception type and message is sufficient to guide test generation in our experiments.

3.3.6 Computing the Fitness of a Test Case with Respect to σ. From a single test case, the client program may invoke *VF* multiple times. To compute the fitness score of a test case, Transfer takes the highest similarity obtained among the multiple invocations of *VF*. Each invocation of *VF* is associated with a set of inputs and outputs. The similarity of an invocation, *inv*, to σ is computed based on *similarity*(*actuals*, *references*), where *actuals* are the inputs and output associated with *inv*. In turn, the test case is as fit as the similarity of the invocation most similar to σ:

$$similarity(inv, references) = similarity(actuals, references)$$
$$fitness(testcase) = max(similarity(inv)),\ inv \in invocations$$

3.4 Implementation

We implement our tool, Transfer, that realizes our novel framework of test mimicry, on top of the implementation of the genetic algorithm and other infrastructure of EvoSuite 1.1.0. Transfer uses the DYNAMOSA [54] evolutionary algorithm for generating test cases due to its suitability for our work and its strong performance over other search algorithms in generating test cases [21, 54]. For evaluating the design decisions unique to Transfer, Transfer uses the default configuration of EvoSuite as prior work has shown that tuning EvoSuite's parameters does not lead to improved performance over its default configuration [14]. The size of the population of test cases is 50 individuals. The default crossover operator is used, which is the single-point crossover with probability of 0.75. Test cases are selected using tournament selection, with a tournament size of 10. Similar to EvoSuite, we use ASM [20] to instrument the code, in particular, the vulnerable function *VF*.

Algorithm 4: *similarity*(*actual*, *reference*, *inspector*): Computing the similarity score of an object to a reference object with respect to a given inspector function

```
Inputs: 1. $actual$, an object from $actuals$
        2. $reference$, an object from $references$
        3. $inspector$, a single inspector function
Output: $similarity$, between 0 and 1. 1 indicates that the
        two objects are equivalent wrt to $inspector$.
function $similarity(actual, reference, inspector)$:
1  $actual\_ins \leftarrow inspector(actual)$
2  $reference\_ins \leftarrow inspector(reference)$
3  if $inspector$ returns a string-like value then
4      return similarity_stringlike($actual\_ins$, $reference\_ins$)
5  end
6  if $inspector$ returns a primitive value then
7      return similarity_primitive($actual\_ins$, $reference\_ins$)
8  end
9  if $inspector$ returns another object then
10     return similarity($actual\_ins$, $reference\_ins$)
11 end
```

4 EMPIRICAL EVALUATION

Our experiments aim to answer the following research questions:

RQ1. Can Transfer generate exploits in client programs that demonstrate library vulnerabilities?

This research question is concerned with the efficacy of Transfer in generating test cases that reveal client usage of vulnerable library functions. Using a benchmark of library vulnerabilities and code from client projects that we have identified by hand, we compute the number of true positives and compute its accuracy. We use Siege [40] as the baseline for comparison.

RQ2. How do the different search goals affect the effectiveness of Transfer?

Transfer has two types of search goals: the primary search goal of satisfying the triggering conditions, and line coverage goals for directing test generation towards the vulnerable library function. We investigate the impact of the goal types on the effectiveness of Transfer via an ablation study.

4.1 Experimental Setup

Experimental subjects. In our experiments, we analyzed 22 recent vulnerabilities from 18 libraries. We manually selected the vulnerabilities for their diversity; the vulnerabilities manifest a range of behaviors, ranging from timeouts and out-of-memory errors (crash behaviors) to incorrect functional behaviors (non-crash behaviors). Similarly, we selected widely-used libraries that cover a range of domains, from file compression (Junrar), parsing XML (Jackson, XStream) and JSON (Json-Smart, OSWAP Json), cryptographic libraries (Bouncy Castle), to HTTP requests (HttpClient). Consequently, the vulnerabilities are not restricted to a single domain. Apart from considering vulnerabilities publicly disclosed as CVEs in the NVD database, we studied several vulnerabilities that

Table 1: The vulnerabilities used in our experiments, including the names and descriptions of the vulnerable libraries, and the effect of the vulnerability. ✗ indicates that an exploit could not be generated. A single ✓ indicates one exploit for a client program was successfully constructed. Two ✓ indicates that exploits were successfully constructed for both client programs.

Vulnerability	Library	Library Description	Effect of vulnerability	Siege	Transfer
CVE-2020-28052	Bouncy Castle	Cryptography	wrong functional behavior	✗	✓✓
CODEC-134 [2]	Apache Codecs	Encoding (e.g. Base64)	wrong functional behavior	✗	✓✓
CVE-2020-13956	Apache HttpClient	HTTP Requests	wrong functional behavior	✗	✓✓
HTTPCLIENT-1803 [7]	Apache HttpClient	HTTP Requests	wrong functional behavior	✗	✗
CVE-2019-14900	Hibernate	ORM framework	SQL injection	✗	✗
CVE-2020-15250	JUnit	Test Framework	wrong file permissions	✓✓	✓✓
CVE-2021-23899	OWASP JSON Sanitizer	JSON processing	arbitrary code injection	✗	✓
CVE-2020-26217	XStream	XML Serialization	remote code execution	✗	✓✓
CVE-2019-12415	Apache POI	Word/PPT/XLS processing	XXE Injection	✗	✗
CVE-2018-1000632	Dom4J	XML processing	XXE Injection	✗	✓
CVE-2020-10693	Hibernate	ORM framework	bypass input sanitization	✗	✗
CVE-2018-1000873	Jackson	JSON processing	slow performance	✗	✗
CVE-2019-12402	Apache Compress	File Compression	infinite loop	✗	✓
CVE-2018-12418	Junrar	File Compression	infinite loop	✗	✓✓
CVE-2019-10094	Apache Tika	Processing files (e.g. xls, pdf)	infinite loop	✗	✓
TwelveMonkeys-595 [5]	TwelveMonkeys	Image processing	infinite loop	✗	✗
CVE-2020-28491	Jackson	JSON processing	out of memory	✗	✗
CVE-2018-1274	Spring Framework	Web development	out of memory	✗	✗
CVE-2021-27568	Json-smart	JSON parser	exception	✓	✓
Zip4J-263 [6]	Zip4J	Processing ZIP files	exception	✓✓	✓✓
Spring Security-8317 [8]	Spring Framework	Web development	exception	✗	✓✓
CVE-2017-7957	XStream	XML serialization	exception	✗	✓✓
			Total	5	23

were fixed silently [26, 62]. These vulnerabilities were identified from publicly accessible vulnerability databases [9, 10].

The vulnerabilities used in our experiments are provided in Table 1. Half of these vulnerabilities (11 vulnerabilities) result in crashes, exceptions, or timeouts. The other 11 vulnerabilities do result in non-crashing behaviors. For example, two vulnerabilities from Apache HttpClient manifest as incorrect functional behavior (e.g. constructing a URI with an incorrect hostname).

For each vulnerability, we identified the vulnerable library function and a vulnerability-witnessing test case through **manual analysis**. Next, we manually identified up to two vulnerable, real client projects for each library vulnerability. For the 22 vulnerabilities, we investigated a total of 64 client projects with source code available on GitHub. From 64 client projects, we obtained 42 pairs of (vulnerability, client program) where the client program is able to trigger the library vulnerability. In the other 24 cases, we[2] confirmed that the client programs are unable to trigger the vulnerability despite calling the vulnerable function, using this to validate that Transfer does not generate test cases unnecessarily. All pairs that we investigated are provided on the project website[3].

While Siege was evaluated on 11 vulnerabilities by Iannone et al. [40], the experiments involved only toy client programs that were written by hand. As such, the client programs may not reflect actual usage of the libraries. We do not use handwritten experimental programs in our experiments. Instead, we use realistic client programs from open-source repositories on GitHub.

Vulnerabilities may depend on many conditions to be triggered, including environmental factors such as software configuration. These environmental factors differ between vulnerabilities. In our experiments, we do not model environmental factors (e.g. specific configurations of the library or client project) or global variables, although it is possible to expand a vulnerability's triggering conditions to account for them.

Baselines. As baseline, we compare Transfer to Siege [40]. Siege is a test generator that targets a single vulnerable line of code given as input. We use the code provided in the replication package of Siege. To use Siege in our experiments, we identified a line of code for each vulnerability. In the ablation study, after removing both types of goals from Transfer, we compare Transfer against EvoSuite, which optimizes for code coverage in the client program.

Experimental setup. We ran Transfer and Siege for up to 10 minutes. In studies related to test generation, experimental durations range from a few minutes [18, 29, 59, 63] to several days [37]. We selected 10 minutes for the same experimental duration as studies on crash reproduction [63]. Our experiments were run on a machine with 2.3 GHz Dual-Core Intel Core i5 with 8GB of RAM.

To mitigate the effect of randomness, we repeat each run 20 times. We consider that a vulnerability is shown to be exploitable from a client program if a tool (i.e., Transfer or Siege) is able to construct an exploit at least 50% of the time, similar to previous studies on crash reproduction [63].

A true positive ("TP") is the successful generation of a test case for the client program that triggers the library's vulnerability. A false negative ("FN") is the failure to generate a test case triggering

[2]The first and second authors manually investigated the client projects.
[3]https://github.com/soarsmu/transfer

Table 2: Number of true positives, false negatives, and proportion of vulnerabilities detected by Transfer, Transfer without directed test generation, Transfer without the triggering conditions, and a baseline EvoSuite.

Approach	# TP	# FN	% detected
TRANSFER	23	19	55%
– directed generation	20	22	48%
– triggering conditions	4	38	10%
– both (EvoSuite)	4	38	10%

the library's vulnerability when the vulnerability can be triggered. To determine if a test case triggers the vulnerability, we manually inspected and ran them to check that they indeed recreated the conditions that triggers the vulnerabilities. We compute the proportion of vulnerable client programs for which Transfer/Siege successfully generates an exploit.

4.2 Experimental Results

4.2.1 RQ1. Efficacy of Transfer. The client projects and vulnerabilities detected by Transfer are given in Table 1. Transfer confirmed the exploitability of the library's vulnerability in 55% (23 / 42) of the client programs. In contrast, Siege [40] was only able to construct an exploit for 5 client projects (12% of the total vulnerable client programs). Transfer outperforms Siege in 13 cases. This shows that Transfer can generate exploits for client programs, outperforming the state-of-the-art test generator by a large margin (over 40% increase in number of generated exploits).

Among the client programs where Transfer succeeded in generating an exploit, Transfer required an average of 27 generations, and took an average of 41 seconds to produce an exploit. This suggests that Transfer could quickly construct exploits for the vulnerabilities in our experiments.

A desirable quality of Transfer is the absence of false alarms. Therefore, Transfer should not produce a test case when the vulnerability cannot be exploited. Among the 24 cases where the client programs are unable to trigger the vulnerability, we found that Transfer exhibited the correct behavior by not producing test cases. Note that the call graphs computed for these pairs indicate that the vulnerable function could be invoked from the client programs, suggesting that existing call graph-based detectors would incorrectly report false alarms.

Among the total of 17 libraries considered, Transfer is able to trigger the vulnerabilities in 14 libraries. Of the 22 vulnerabilities, 11 are not crash-related and Transfer detects 7 of these 11 vulnerabilities. This indicates that Transfer is able to detect non-crash vulnerabilities. As for the other 11 vulnerabilities with exceptions, crashes, or timeouts, Transfer detects 7 of them, suggesting that Transfer is effective for both crash and non-crash vulnerabilities.

Answer to RQ1: Transfer is able to generate exploits for vulnerabilities for 14 libraries, producing 23 exploits for 64 client programs. The baseline Siege was only able to generate 5 exploits.

4.2.2 RQ2. Ablation Study. Next, we performed an ablation study on Transfer by disabling the search goals accordingly. Table 2 shows the results of this experiment.

Table 3: The average number of generations required (number of seconds given in parentheses) for Transfer to find an exploit, with and without the use of the line coverage goals that direct test generation towards code on the call graph between the client program and the vulnerable function.

Vulnerability	Transfer	–directed test generation
CODEC-134	34 (21s)	20 (15s)
CVE-2020-13956	26 (41s)	58 (50s)
CVE-2021-23899	2 (20s)	6 (108s)
CVE-2019-12402	7 (11s)	343 (328s)

Without the primary goal of the triggering conditions, Transfer created exploits of only 3 vulnerabilities from the client programs. In this setting, Transfer is equivalent to EvoSuite if EvoSuite considers only the line and branch coverage goals of the path along the call graph and the vulnerable function are provided. The decrease in effectiveness shows that the triggering conditions are essential. Without them, Transfer is unable to select a test case that confirms that the vulnerability is exploitable.

By dropping the line coverage goals that direct test generation towards the vulnerable library function, Transfer is unable to trigger the vulnerability in 3 cases (*CVE-2020-28052*, *CVE-2020-13956*, *CVE-2019-10094*). Without the line coverage goals, Transfer had to generate a test case that reached the vulnerable function through completely random mutations. As such, the generated test cases are less likely to invoke the vulnerable function.

Table 3 shows the four cases where we observed that directed test generation substantially changed the number of generations required for Transfer to discover an exploit. For the other cases, we did not notice a significant change in the number of generations required. In 3 of 4 cases, the line coverage goals allowed Transfer to discover the exploit in a smaller number of generations. However, in one case (*CODEC-134*), the line coverage goals slowed down the search. In the other 16 cases, Transfer performed similarly with or without the line coverage goals.

Along with the 3 cases where Transfer could not reveal the vulnerability in the client programs, there are a total of 7 cases (41% of the 17 cases where Transfer could detect the vulnerability in the client program) where the line coverage goals increased the effectiveness of Transfer. Therefore, we see that the line coverage goals were important, although not essential, in our experiments.

Answer to RQ2: While test generation directed by line coverage helped, guiding test generation to satisfy the triggering conditions extracted from the carved program state was essential to Transfer.

5 DISCUSSION

We qualitatively analyse our results to better understand the performance of Transfer and discuss threats to validity.

5.1 Qualitative Analysis

5.1.1 Why did Transfer Work?

Generating complex inputs. Both Transfer and Siege direct test generation towards the vulnerable library code. However, there

```
arg1.getRawAuthority()==
    "blah@goggle.com:80@google.com"
&& arg1.getScheme() == "http"
&& arg1.getRawPath() == "/"
...
```

Figure 6: Part of the triggering conditions identified for CVE-2020-13956

are 18 client programs where Siege was not able to uncover the library vulnerability while Transfer was able to. Our ablation study reveals that capturing the triggering conditions of the vulnerabilities was the key to Transfer's effectiveness. Indeed, many vulnerabilities are corner cases and exceptional behaviors of the software system, and Transfer targets the triggering conditions, which provides better guidance for uncovering the vulnerability.

As an example, we use CVE-2020-13956 from Section 2.3, a simplified version of the triggering conditions captured from the library's test case is shown in Figure 6. The vulnerability manifests when an invalid URL (e.g. "blah@goggle.com:80@google.com") is passed into the vulnerable function, and causes a connection to an unexpected host. Neither EvoSuite nor Siege account for the necessary inputs and program state for the exploit. EvoSuite aims to covers all other behaviors, while Transfer targets a single behavior encapsulated by the triggering conditions, σ. Siege has to construct an invalid URL from scratch, while Transfer leverages the input from the library's test case.

Guidance from triggering conditions. In Table 2, Transfer using only directed test generation without the triggering conditions performs identically to EvoSuite. Given that only one class is targeted, the search budget of 10 minutes may have provided enough time for EvoSuite to cover the same code locations that Transfer was directed to. This indicates that the improvements of Transfer over EvoSuite observed in our experiments stem from the use of the triggering conditions.

Compared to Siege, Transfer and Siege fundamentally differ in how they guide the test generator. Siege checks for the coverage of a vulnerable line of code in the library, while Transfer checks if program state reached by the library test case has been reproduced. In 7 vulnerabilities in our experiments, both benign and malicious inputs would cover the same lines of code, and, hence, Siege does not succeed in creating an exploit. In these cases, guiding the test generator by code coverage may not be enough for producing an exploit.

5.1.2 *Complementary to Existing Tools.*

We suggest test mimicry is complementary to existing, call graph-based techniques [32, 58]. Call graph-based techniques produce false positives, which may lead to low adoption [41, 45]. If Transfer succeeds in demonstrating that a vulnerability can be exploited, then developer effort can be reduced. However, if Transfer fails in generating a test case, it does not mean that the vulnerability is not exploitable, and developers will have to expend effort in investigating the vulnerability.

Test mimicry relies on a test case from the library. While the test case demonstrates the vulnerability, the extracted triggering conditions, σ, do not characterize all possible triggers of the vulnerability. For example, the test case of CVE-2020-13956 (Figure 6) shows one possible URL out of many that could trigger the vulnerability. Nevertheless, as our goal is to demonstrate that the vulnerability may be exploited from a client program, having a single example may suffice to guide the creation of a test case demonstrating the vulnerability's exploitability.

5.1.3 *Next Steps.*

While Transfer was successful in a number of cases in our evaluation, we identified some limitations to be addressed in future work.

Dependence on inspector functions. In *TwelveMonkeys-595*, Transfer is unable to produce a test case as the arguments to the vulnerable function lacked inspector functions. Transfer could not extract triggering conditions that capture the program state. In the future, we plan to additionally use other means of extracting vulnerability-triggering conditions beyond the use of inspectors.

Irrelevant program states. In Figure 6, the triggering conditions captured aspects that were irrelevant to triggering the vulnerability (e.g. `getScheme() ==` "http") along with the necessary aspects (having `getRawAuthority()` set to an invalid URL, "blah@goggle.com:80@google.com"). In other words, the triggering conditions captured by Transfer are more specific than the actual conditions required to trigger the vulnerability. While Transfer is able to construct a test case that satisfies the triggering conditions in Figure 6, including its irrelevant aspects, we found that in other cases, Transfer may have been limited by its inability to satisfy the irrelevant aspects of the captured triggering conditions. In the future, we will explore techniques from the area of test input minimization for the extracted triggering conditions.

5.2 Threats to Validity

A threat to validity is the manual selection of vulnerabilities in our experiments. While the type of library test case is not a limitation of our approach, we selected vulnerabilities with test cases that target only the vulnerable behavior (i.e., the test case does not exercise unrelated behaviors of the vulnerable code location). Otherwise, Transfer has no way to correctly identify the right behavior associated with the vulnerability.

Our experiments focused on the effectiveness of the test mimicry technique. As such, we focused on evaluating the design decisions unique to Transfer and used the default parameters of EvoSuite. If we fine-tune the parameters of the search parameters, Transfer may perform better in our experiments and we leave these experiments for future work. Likewise, we used the DynaMOSA search algorithm as it has been shown to outperform other search algorithms in generating test cases [21, 54].

Regarding the generalizability of our findings, we selected vulnerabilities that range across multiple domains for our experiments. Table 4 shows the number of successfully generated test cases for the domains of libraries considered in our experiments. As our experiments covered various domains, we believe our approach is not limited to specific domains and we expect it to generalize to domains beyond our evaluation. Still, it is possible that it will not work on some domains that we have not identified yet. Compared to the benchmark of Iannone et al. [40], our benchmark is larger and uses realistic client programs instead of toy programs.

Table 4: The domains of the libraries considered and the number of generated exploits in our experiments.

Domain	Libraries	# exploits
File compression	Zip4J, Commons Compress, Junrar	5
Serialization/deserialization	XStream, Json-smart, jackson, dom4j	6
Web development/utilities	Spring, Hibernate ORM, Apache HttpComponents	4
Data encoding/cryptography	Bouncy castle, Apache Codecs, Apache POI	4
File formats (PDFs, images)	Apache Tika, Twelvemonkey	1
Test Framework	JUnit	1

6 RELATED WORK

Software Composition Analysis and Search-based Test Generation are discussed in Section 2. Here, we discuss other related works.

Automated Exploit Generation. Our work is related to the field of Automated Exploit Generation [15, 22], with the same goal of producing exploits for known vulnerabilities. Brumley et al. [19] have shown the possibility of generating exploits based on the patch fixing the vulnerability. While our work relies on information in a patch, we demonstrate that the test case accompanying the bug fix can be used to generate exploits, while prior work used only the modified code. Moreover, Brumley et al. [19] only creates exploits of vulnerabilities associated with missing input validation.

Other studies that automatically generate exploits study specific types of vulnerabilities for a narrow range of software systems [25, 36, 39, 67–69]. For example, Chen et al. [25] generate exploits for vulnerabilities caused by out-of-bounds writes, generating 11 exploits. You et al. [69] proposes the use of natural language processing to extract information to guide fuzzing, triggering 18 vulnerabilities in the Linux kernel. In contrast, TRANSFER construct test cases for a wide range of vulnerabilities for Java programs.

Test input generation. There is a large amount of work on test input generation [1, 17, 18, 24, 34, 37, 38, 42, 47, 49, 53, 64]. These studies use fuzzing or symbolic execution, focus on maximizing coverage [1, 18, 24, 34, 53], or finding vulnerabilities with oracles such as crashes or poor performance [47, 48, 57]. Compared to these studies, apart from generating test inputs (e.g. a string), TRANSFER constructs the sequence of different function calls (e.g. constructors, setter functions) that set up the object states necessary to trigger the vulnerability. Provided with a vulnerability-witnessing test case, TRANSFER focuses on demonstrating a single, vulnerable behavior and is able to detect vulnerabilities that do not manifest as crashes.

Directed fuzzing [17, 23, 66] focuses fuzzing efforts on selected code locations. Similarly, TRANSFER directs test generation towards the vulnerable code location. However, beyond reaching the same code locations, TRANSFER focuses on reproducing the program state reached in a library test case to check if a library vulnerability can be exploited from a client program.

Crash reproduction. Botsing [29] is an approach that uses search-based test generation to reproduce crashes given stacktraces. Botsing is complementary to our approach in detecting vulnerabilities that result in crashes. However, not all vulnerabilities manifest as exceptions and Botsing cannot be used without a stacktrace. As such, we do not use Botsing in our experiments as just 4 out of the 22 studied vulnerabilities result in exceptions.

Reusing test cases. Researchers have proposed techniques that reuses existing test cases from other projects [70]. Given similar but not identical programs, Zhang and Kim [70] try to reuse existing test cases to enable behavioral comparison between the programs. Our work is similar in reusing properties of an existing test case in a different software system, but we focus on generating tests for client programs rather than similar programs. While Mujahid et al. [52] proposes the use of client test cases to detect breaking changes in libraries, our work proposes the use of library test cases to detect the exploitability of library vulnerabilities from client programs.

7 CONCLUSION AND FUTURE WORK

We propose *test mimicry* to leverage vulnerability-witnessing test cases from open-source libraries, generating test cases for client projects that demonstrate the exploitability of the library vulnerabilities. Our tool, TRANSFER, captures the program states reached by the library test cases, directing test case generation for client programs towards reconstructing them. TRANSFER generates 4x more exploits for real client programs than the state-of-the-art approach.

In the future, we will explore more expressive conditions to represent the program state, including incorporating facts about the environment (e.g. project configuration). We also wish to explore techniques to minimize the triggering conditions to remove aspects irrelevant to the vulnerability.

Acknowledgement. This project is supported by the National Research Foundation, Singapore and National University of Singapore through its National Satellite of Excellence in Trustworthy Software Systems (NSOE-TSS) office under the Trustworthy Computing for Secure Smart Nation Grant (TCSSNG) award no. NSOE-TSS2020-02, and the Australian Government through the Australian Research Council's Discovery Early Career Researcher Award, project number DE220101057. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore and National University of Singapore (including its National Satellite of Excellence in Trustworthy Software Systems (NSOE-TSS) office).

REFERENCES

[1] [n.d.]. American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/technical_details.txt.
[2] [n.d.]. CODEC-134 from Apache Commons Codecs's issue tracker. https://issues.apache.org/jira/browse/CODEC-134.
[3] [n.d.]. CVE-2019-12402. https://nvd.nist.gov/vuln/detail/CVE-2019-12402.
[4] [n.d.]. Google Security Blog: Understanding the Impact of Apache Log4j Vulnerability. https://security.googleblog.com/2021/12/understanding-impact-of-apache-log4j.html.
[5] [n.d.]. SNYK-JAVA-COMTWELVEMONKEYSIMAGEIO-1083830 from SNYK. https://snyk.io/vuln/SNYK-JAVA-COMTWELVEMONKEYSIMAGEIO-1083830.
[6] [n.d.]. SNYK-JAVA-NETLINGALAZIP4J-1074967 from SNYK. https://snyk.io/vuln/SNYK-JAVA-NETLINGALAZIP4J-1074967.

[7] [n.d.]. SNYK-JAVA-ORGAPACHEHTTPCOMPONENTS-31517 from SNYK. https://snyk.io/vuln/SNYK-JAVA-ORGAPACHEHTTPCOMPONENTS-31517.
[8] [n.d.]. SNYK-JAVA-ORGSPRINGFRAMEWORKSECURITY-570204 from SNYK. https://snyk.io/vuln/SNYK-JAVA-ORGSPRINGFRAMEWORKSECURITY-570204.
[9] [n.d.]. Snyk's vulnerability database. https://snyk.io/vuln?type=maven.
[10] [n.d.]. SourceClear's vulnerability database. https://www.sourceclear.com/vulnerability-database/.
[11] [n.d.]. Wired: The Log4J Vulnerability Will Haunt the Internet for Years. https://www.wired.com/story/log4j-log4shell/.
[12] M Moein Almasi, Hadi Hemmati, Gordon Fraser, Andrea Arcuri, and Janis Benefelds. 2017. An industrial evaluation of unit test generation: Finding real faults in a financial application. In *IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. IEEE, 263–272. https://doi.org/10.1109/ICSE-SEIP.2017.27
[13] Andrea Arcuri. 2013. It really does matter how you normalize the branch distance in search-based software testing. *Software Testing, Verification and Reliability* 23, 2 (2013), 119–147. https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.457
[14] 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. https://doi.org/10.1007/s10664-013-9249-9
[15] Thanassis Avgerinos, Sang Kil Cha, Alexandre Rebert, Edward J Schwartz, Maverick Woo, and David Brumley. 2014. Automatic exploit generation. *Commun. ACM* 57, 2 (2014), 74–84. https://doi.org/10.1145/2560217.2560219
[16] Kent Beck. 2003. *Test-driven development: by example.* Addison-Wesley Professional.
[17] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed greybox fuzzing. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2329–2344. https://doi.org/10.1145/3133956.3134020
[18] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2017. Coverage-based greybox fuzzing as Markov Chain. *IEEE Transactions on Software Engineering (TSE)* 45, 5 (2017), 489–506. https://doi.org/10.1109/TSE.2017.2785841
[19] David Brumley, Pongsin Poosankam, Dawn Song, and Jiang Zheng. 2008. Automatic patch-based exploit generation is possible: Techniques and implications. In *IEEE Symposium on Security and Privacy (S&P)*. IEEE, 143–157. https://doi.org/10.1109/SP.2008.17
[20] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. 2002. ASM: a code manipulation tool to implement adaptable systems. *Adaptable and extensible component systems* 30, 19 (2002).
[21] 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 (IST)* 104 (2018), 207–235. https://doi.org/10.1016/j.infsof.2018.08.010
[22] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. 2012. Unleashing Mayhem on binary code. In *IEEE Symposium on Security and Privacy (S&P)*. IEEE, 380–394. https://doi.org/10.1109/SP.2012.31
[23] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. 2018. Hawkeye: Towards a desired directed grey-box fuzzer. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2095–2108. https://doi.org/10.1145/3243734.3243849
[24] Peng Chen and Hao Chen. 2018. Angora: Efficient fuzzing by principled search. In *IEEE Symposium on Security and Privacy (S&P)*. IEEE, 711–725. https://doi.org/10.1109/SP.2018.00046
[25] Weiteng Chen, Xiaochen Zou, Guoren Li, and Zhiyun Qian. 2020. KOOBE: Towards Facilitating Exploit Generation of Kernel Out-Of-Bounds Write Vulnerabilities. In *USENIX Security Symposium (USENIX Security)*. 1093–1110. https://www.usenix.org/conference/usenixsecurity20/presentation/chen-weiteng
[26] Yang Chen, Andrew E Santosa, Ang Ming Yi, Abhishek Sharma, Asankhaya Sharma, and David Lo. 2020. A Machine Learning Approach for Vulnerability Curation. In *International Conference on Mining Software Repositories (MSR)*. 32–42. https://doi.org/10.1145/3379597.3387461
[27] Valentin Dallmeier, Christian Lindig, Andrzej Wasylkowski, and Andreas Zeller. 2006. Mining object behavior with ADABU. In *Proceedings of the 2006 international workshop on Dynamic systems analysis*. 17–24. https://dl.acm.org/doi/10.1145/1138912.1138918
[28] Alexandre Decan, Tom Mens, and Eleni Constantinou. 2018. On the impact of security vulnerabilities in the npm package dependency network. In *International Conference on Mining Software Repositories (MSR)*. 181–191. https://doi.org/10.1145/3196398.3196401
[29] Pouria Derakhshanfar, Xavier Devroey, Annibale Panichella, Andy Zaidman, and Arie van Deursen. 2020. Botsing, a Search-based Crash Reproduction Framework for Java. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1278–1282. https://doi.org/10.1145/3324884.3415299
[30] Xavier Devroey, Sebastiano Panichella, and Alessio Gambi. 2020. Java Unit Testing Tool Competition: Eighth Round. In *IEEE/ACM International Conference on Software Engineering Workshops*. 545–548. https://doi.org/10.1145/3387940.3392265
[31] Sebastian Elbaum, Hui Nee Chin, Matthew B Dwyer, and Jonathan Dokulil. 2006. Carving differential unit test cases from system test cases. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. 253–264. https://doi.org/10.1145/1181775.1181806
[32] Darius Foo, Jason Yeo, Hao Xiao, and Asankhaya Sharma. 2019. The dynamics of Software Composition Analysis. *Automated Software Engineering (ASE) (Late Breaking Results)* (2019). http://arxiv.org/abs/1909.00973
[33] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: automatic test suite generation for object-oriented software. In *ACM SIGSOFT Symposium and the European Conference on Foundations of Software Engineering (FSE)*. 416–419. https://doi.org/10.1145/2025113.2025179
[34] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. 2018. CollAFL: Path sensitive fuzzing. In *IEEE Symposium on Security and Privacy (S&P)*. IEEE, 679–696. https://doi.org/10.1109/SP.2018.00040
[35] David Grove and Craig Chambers. 2001. A framework for call graph construction algorithms. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 23, 6 (2001), 685–746. https://doi.org/10.1145/506315.506316
[36] Sean Heelan, Tom Melham, and Daniel Kroening. 2019. Gollum: Modular and greybox exploit generation for heap overflows in interpreters. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 1689–1706. https://doi.org/10.1145/3319535.3354224
[37] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with code fragments. In *USENIX Security Symposium (USENIX Security)*. 445–458. https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/holler
[38] Matthias Hoschele and Andreas Zeller. 2017. Mining input grammars with AUTOGRAM. In *IEEE/ACM International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 31–34. https://doi.org/10.1109/ICSE-C.2017.14
[39] Hong Hu, Zheng Leong Chua, Sendroiu Adrian, Prateek Saxena, and Zhenkai Liang. 2015. Automatic generation of data-oriented exploits. In *USENIX Security Symposium (USENIX Security)*. 177–192. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/hu
[40] Emanuele Iannone, Dario Di Nucci, Antonino Sabetta, and Andrea De Lucia. 2021. Toward Automated Exploit Generation for Known Vulnerabilities in Open-Source Libraries. In *IEEE/ACM International Conference on Program Comprehension (ICPC)*. IEEE, 396–400. https://doi.org/10.1109/ICPC52881.2021.00046
[41] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why don't software developers use static analysis tools to find bugs?. In *IEEE/ACM International Conference on Software Engineering (ICSE)*. IEEE, 672–681. https://doi.org/10.1109/ICSE.2013.6606613
[42] Alexander Kampmann, Nikolas Havrikov, Ezekiel O Soremekun, and Andreas Zeller. 2020. When does my program do this? learning circumstances of software behavior. In *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 1228–1239. https://doi.org/10.1145/3368089.3409687
[43] Pavneet Singh Kochhar, Tegawendé F Bissyandé, David Lo, and Lingxiao Jiang. 2013. Adoption of software testing in open source projects–A preliminary study on 50,000 projects. In *2013 European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE, 353–356. https://doi.org/10.1109/CSMR.2013.48
[44] Pavneet Singh Kochhar, Ferdian Thung, and David Lo. 2015. Code coverage and test suite effectiveness: Empirical study with real bugs in large systems. In *IEEE International conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 560–564. https://doi.org/10.1109/SANER.2015.7081877
[45] Pavneet Singh Kochhar, Xin Xia, David Lo, and Shanping Li. 2016. Practitioners' expectations on automated fault localization. In *International Symposium on Software Testing and Analysis*. 165–176. https://doi.org/10.1145/2931037.2931051
[46] Raula Gaikovina Kula, Daniel M German, Ali Ouni, Takashi Ishio, and Katsuro Inoue. 2018. Do developers update their library dependencies? *Empirical Software Engineering* 23, 1 (2018), 384–417. https://doi.org/10.1007/s10664-017-9521-5
[47] Xuan-Bach D Le, Corina Pasareanu, Rohan Padhye, David Lo, Willem Visser, and Koushik Sen. 2019. SAFFRON: Adaptive grammar-based fuzzing for worst-case analysis. *ACM SIGSOFT Software Engineering Notes* 44, 4 (2019), 14–14. https://doi.org/10.1145/3364452.3364455
[48] Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. 2018. PerfFuzz: Automatically generating pathological inputs. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 254–265. https://doi.org/10.1145/3213846.3213874
[49] Björn Mathis, Rahul Gopinath, and Andreas Zeller. 2020. Learning input tokens for effective fuzzing. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 27–37. https://doi.org/10.1145/3395363.3397348
[50] Phil McMinn. 2004. Search-based software test data generation: a survey. *Software testing, Verification and reliability* 14, 2 (2004), 105–156. https://doi.org/10.1002/stvr.294
[51] Samim Mirhosseini and Chris Parnin. 2017. Can automated pull requests encourage software developers to upgrade out-of-date dependencies?. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 84–94. https://doi.org/10.1109/ASE.2017.8115621
[52] Suhaib Mujahid, Rabe Abdalkareem, Emad Shihab, and Shane McIntosh. 2020. Using Others' Tests to Identify Breaking Updates. In *International Conference on Mining Software Repositories (MSR)*. 466–476. https://doi.org/10.1145/

3379597.3387476
[53] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. 2019. Semantic fuzzing with Zest. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 329–340. https://doi.org/10.1145/3293882.3330576
[54] Annibale Panichella, Fitsum Meshesha 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 (TSE)* 44, 2 (2017), 122–158. https://doi.org/10.1109/TSE.2017.2663435
[55] Ivan Pashchenko, Henrik Plate, Serena Elisa Ponta, Antonino Sabetta, and Fabio Massacci. 2018. Vulnerable open source dependencies: Counting those that matter. In *ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 1–10. https://doi.org/10.1145/3239235.3268920
[56] Ivan Pashchenko, Henrik Plate, Serena Elisa Ponta, Antonino Sabetta, and Fabio Massacci. 2020. Vuln4real: A methodology for counting actually vulnerable dependencies. *IEEE Transactions on Software Engineering (TSE)* (2020). https://ieeexplore.ieee.org/abstract/document/9201023/
[57] Theofilos Petsios, Jason Zhao, Angelos D Keromytis, and Suman Jana. 2017. SlowFuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2155–2168. https://doi.org/10.1145/3133956.3134073
[58] Serena Elisa Ponta, Henrik Plate, and Antonino Sabetta. 2018. Beyond metadata: Code-centric and usage-based analysis of known vulnerabilities in open-source software. In *IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 449–460. https://doi.org/10.1109/ICSME.2018.00054
[59] Sameer Reddy, Caroline Lemieux, Rohan Padhye, and Koushik Sen. 2020. Quickly generating diverse valid test inputs with Reinforcement Learning. In *IEEE/ACM International Conference on Software Engineering (ICSE)*. IEEE, 1410–1421. https://doi.org/10.1145/3377811.3380399
[60] José Miguel Rojas, Gordon Fraser, and Andrea Arcuri. 2016. Seeding strategies in search-based unit test generation. *Software Testing, Verification and Reliability* 26, 5 (2016), 366–401. https://doi.org/10.1002/stvr.1601
[61] Atanas Rountev, Scott Kagan, and Michael Gibas. 2004. Static and dynamic analysis of call chains in Java. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 1–11. https://doi.org/10.1145/1007512.1007514
[62] Antonino Sabetta and Michele Bezzi. 2018. A practical approach to the automatic classification of security-relevant commits. In *IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 579–582. https://doi.org/10.1109/ICSME.2018.00058
[63] Mozhan Soltani, Annibale Panichella, and Arie Van Deursen. 2018. Search-based crash reproduction and its impact on debugging. *IEEE Transactions on Software Engineering (TSE)* 46, 12 (2018), 1294–1317. https://doi.org/10.1109/TSE.2018.2877664
[64] Willem Visser, Corina S Păsăreanu, and Sarfraz Khurshid. 2004. Test input generation with Java PathFinder. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 97–107. https://doi.org/10.1145/1007512.1007526
[65] Ying Wang, Bihuan Chen, Kaifeng Huang, Bowen Shi, Congying Xu, Xin Peng, Yijian Wu, and Yang Liu. 2020. An empirical study of usages, updates and risks of third-party libraries in Java projects. In *IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 35–45. https://doi.org/10.1109/ICSME46990.2020.00014
[66] Yunchao Wang, Zehui Wu, Qiang Wei, and Qingxian Wang. 2019. NeuFuzz: Efficient fuzzing with deep neural network. *IEEE Access* 7 (2019), 36340–36352. https://doi.org/10.1109/ACCESS.2019.2903291
[67] Yan Wang, Chao Zhang, Zixuan Zhao, Bolun Zhang, Xiaorui Gong, and Wei Zou. 2021. MAZE: Towards Automated Heap Feng Shui. In *USENIX Security Symposium (USENIX Security)*. https://www.usenix.org/conference/usenixsecurity21/presentation/wang-yan
[68] Wei Wu, Yueqi Chen, Jun Xu, Xinyu Xing, Xiaorui Gong, and Wei Zou. 2018. FUZE: Towards facilitating exploit generation for kernel use-after-free vulnerabilities. In *USENIX Security Symposium (USENIX Security)*. 781–797. https://www.usenix.org/conference/usenixsecurity18/presentation/wu-wei
[69] Wei You, Peiyuan Zong, Kai Chen, XiaoFeng Wang, Xiaojing Liao, Pan Bian, and Bin Liang. 2017. SemFuzz: Semantics-based automatic generation of proof-of-concept exploits. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2139–2154. https://doi.org/10.1145/3133956.3134085
[70] Tianyi Zhang and Miryung Kim. 2017. Automated transplantation and differential testing for clones. In *IEEE/ACM International Conference on Software Engineering (ICSE)*. IEEE/ACM, 665–676. https://doi.org/10.1109/ICSE.2017.67