4-2024

Exploiting library vulnerability via migration-based automated test generation

Zirui CHEN

Xing HU

Xin XIA

Yi GAO

Tongtong XU

See next page for additional authors

Citation

CHEN, Zirui; HU, Xing; XIA, Xin; GAO, Yi; XU, Tongtong; LO, David; and YANG, Xiaohu. Exploiting library vulnerability via migration-based automated test generation. (2024). *ICSE '24: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, Lisbon, Portugal, April 14-20*. 1-12.
Available at: https://ink.library.smu.edu.sg/sis_research/9253

Exploiting Library Vulnerability via Migration Based Automating Test Generation

Zirui Chen
Zhejiang University
China
chenzirui@zju.edu.cn

Xing Hu*
Zhejiang University
China
xinghu@zju.edu.cn

Xin Xia
Huawei
China
xin.xia@acm.org

Yi Gao
Zhejiang University
China
gaoyi01@zju.edu.cn

Tongtong Xu
Huawei
China
xutongtong9@huawei.com

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

Xiaohu Yang
Zhejiang University
China
yangxh@zju.edu.cn

ABSTRACT

In software development, developers extensively utilize third-party libraries to avoid implementing existing functionalities. When a new third-party library vulnerability is disclosed, project maintainers need to determine whether their projects are affected by the vulnerability, which requires developers to invest substantial effort in assessment. However, existing tools face a series of issues: static analysis tools produce false alarms, dynamic analysis tools require existing tests and test generation tools have low success rates when facing complex vulnerabilities.

Vulnerability exploits, as code snippets provided for reproducing vulnerabilities after disclosure, contain a wealth of vulnerability-related information. This study proposes a new method based on vulnerability exploits, called Vesta (Vulnerability Exploit-based Software Testing Auto-Generator), which provides vulnerability exploit tests as the basis for developers to decide whether to update dependencies. Vesta extends the search-based test generation methods by adding a migration step, ensuring the similarity between the generated test and the vulnerability exploit, which increases the likelihood of detecting potential library vulnerabilities in a project.

We perform experiments on 30 vulnerabilities disclosed in the past five years, involving 60 vulnerability-project pairs, and compare the experimental results with the baseline method, Transfer. The success rate of Vesta is 71.7% which is a 53.4% improvement over Transfer in the effectiveness of verifying exploitable vulnerabilities.

*Corresponding Author

ICSE 2024, April 2024, Lisbon, Portugal

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

KEYWORDS

Library Vulnerabilities, Search-based Test Generation

ACM Reference Format:
Zirui Chen, Xing Hu, Xin Xia, Yi Gao, Tongtong Xu, David Lo, and Xiaohu Yang. 2023. Exploiting Library Vulnerability via Migration Based Automating Test Generation . In *Proceedings of 46th International Conference on Software Engineering (ICSE 2024).* ACM, New York, NY, USA, 12 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Open-source libraries are widely used during software development [31, 45]. It is estimated that 96% of software projects contain open source resources [42]. The widespread use of open-source libraries allows developers to reuse common functionalities, saving time and resources [24, 31, 48]. Similar to other software projects, open-source libraries may contain flaws [8, 12], which increase the possibility of attacks on software systems [11, 43]. Vulnerabilities in open-source libraries can be particularly serious for the following reasons: 1) The vulnerabilities may propagate to dependent packages [15, 34]; 2) Vulnerabilities in open-source libraries can expose client applications to abuse [26]. For example, in 2021, a remote code execution bug was disclosed in the log4j2 library, which affected millions of devices and required developers to upgrade dependency quickly [27, 46].

When a new fix for existing bugs in a library is released, developers must decide if the dependency version should be updated. However, updating the dependency version may introduce conflicts into software ecosystems [9]. For instance, the library API may break when fixing bugs, refactoring code, or adding features [28, 45], making it impossible for developers to upgrade the dependency version immediately. Due to this reason, a significant amount of projects are exposed to the library vulnerabilities [14, 18, 35, 45, 49].

The inclusion of a vulnerable dependency in a project does not necessarily mean that the project is affected by the vulnerability [48]. According to Zapata's research [19], 73.3% of projects that

depend on vulnerable dependencies are secure. For a library vulnerability to be exploitable, it must satisfy two conditions: 1) the project must contain a control flow that calls the vulnerable function, and 2) the client project should be able to pass a crafted input that triggers the vulnerability [3]. For this reason, developers should check whether the project is affected by the vulnerability before updating the dependency version. This task can be particularly time-consuming in large projects [13].

To release developers from checking the vulnerabilities' exploitability in projects, existing tools are developed to detect whether the project is affected by the vulnerability. Dependency-based methods analyze vulnerable dependency versions to check potential vulnerabilities [2, 15, 19]. Call graph-based methods analyze the control flow of the project, checking if it contains a function call to the vulnerable function [36, 38]. Dynamic detecting methods executing existing tests or generated tests, checking whether a control flow could reach the vulnerable function to detect the exploitable vulnerabilities [21, 39].

However, the aforementioned methods will cause false alarms [10] as the lack of measuring whether client projects could construct inputs to trigger the vulnerabilities. To solve the problem, some researchers [26, 27] focused on generating a test case to call the project APIs as attackers. The test could be an exploit if it could trigger the library vulnerability in the project. Recently, Transfer [27] utilized vulnerability witness tests to overcome the lack of domain knowledge and the intrinsic complexity of exploiting vulnerabilities. In Transfer, vulnerability witness tests are used to collect the trigger condition. By carving the libraries' test, Transfer obtained the program state associated with the triggering of the vulnerability, which is added to the fitness function to evaluate the possibility of exploiting the library vulnerability.

Similarly, we use vulnerability exploit code to collect domain knowledge for triggering. We collect a total of 747 projects that rely on the Jackson-Databind library, which is used to read content encoded in JSON or other data formats as well, as long as the parser and generator implementations exist [20]. Jackson-Databind is associated with CVE-2019-14540 and an additional 46 vulnerabilities. We specifically chose 247 projects that include a call graph for the vulnerable function called `readValue`. Through manual examination of these call graphs, we discover that certain projects either passed parameters to the vulnerable function without making any alterations or made simple modifications, such as changing the parameter type or adding a substring. Based on the aforementioned conclusion, we propose a hypothesis that transferring parameter values extracted from vulnerability exploit to a specific call graph enables the vulnerable functions in the project to receive a parameter value capable of triggering the vulnerability.

In this paper, we propose Vesta to assess the exploitability of vulnerabilities by generating test cases that trigger the vulnerabilities. Instead of executing the witness test in Transfer, we collect parameters from the exploit to ensure the trigger of vulnerabilities. For each vulnerability, we collect an exploit code snippet that provides domain knowledge for triggering the vulnerability. Vesta extracts parameters from the exploit code and utilizes rules to modify it. Finally, we migrate the modified parameters into the generated tests, resulting in a 63.3% improvement in performance.

We use static analysis tools (e.g., javacg-static) to locate the entry function and vulnerable function, which guides generating a high test coverage. If the test case successfully reproduces the vulnerability after incorporating the parameters, the project is deemed to have an exploitable library vulnerability, and the test case serves as the exploit in the project. Additionally, though projects' existing tests are not effective in detecting potential library vulnerabilities [26], they provide extensive information related to the function call in the project. If existing tests cover the vulnerable function, we can migrate parameters into these tests instead of generating tests.

We evaluate Vesta and our baseline, Transfer, by using a dataset consisting of 30 vulnerabilities and 60 projects affected by library vulnerabilities sourced from GitHub. Vesta successfully generates 43 exploits for 26 vulnerabilities, outperforming Transfer, which produces 11 exploits. On average, it takes 22.75 seconds to generate an exploit when no existing test case is available. Additionally, we test our method in eight projects with tests covering vulnerable functions. By migrating parameter values into existing tests, all of the projects' library vulnerabilities are exploited.

In summary, we make the following contributions:

- We propose a migration-based test generation method to provide domain knowledge for triggering the vulnerability. Our tool is available on our website[1].
- We implement an exploit generator that exploits library vulnerabilities by utilizing the vulnerable function's position within the project and domain knowledge provided by library exploit code.
- We evaluate our method on 30 known vulnerabilities across 60 Java vulnerability-project pairs, resulting in the successful generation of 32 additional exploits compared to baseline Transfer.

The remainder of the paper is organized as follows. In Section 2, we present a motivating example. In Section 3, we describe the implementation details of our method. In Section 4, we demonstrate the effectiveness of our approach through empirical evaluation. In Section 5, we discuss the reliability of our method. In Section 6, we provide an overview of the related work in this paper. Finally, in Section 7, we summarize our method and mention future works.

2 MOTIVATION EXAMPLE

Figure 1 presents the description of CVE-2023-1370, which provides information about the vulnerability, e.g. the root cause and the reproduction steps of the vulnerability. However, the description only mentions that a processed input stream will result in stack overflow without providing specific information about the details of the input. The lack of details makes it difficult for developers to determine whether the project has been affected as developers don't know which input will cause the problem.

As shown in Figure 2, in some security research websites [2], we can find the vulnerability exploit (POC) for CVE-2023-1370. The exploit initially constructs a string with specific characteristics to trigger the vulnerability, which uses two `for` loops to make a nested object. Subsequently, this value is passed to the vulnerable function. Executing this exploit code on a vulnerable version of the library will result in a denial of service issue. This exploit reveals that if the vulnerable function receives a value similar to *s*, the

[1] https://github.com/chen-zirui/TestMigration
[2] https://research.jfrog.com/

CVE-2023-1370 Detail

Description

[Json-smart](https://netplex.github.io/json-smart/) is a performance focused, JSON processor lib. When reaching a '[' or '{' character in the JSON input, the code parses an array or an object respectively. It was discovered that the code does not have any limit to the nesting of such arrays or objects. Since the parsing of nested arrays and objects is done recursively, nesting too many of them can cause a stack exhaustion (stack overflow) and crash the software.

Figure 1: Description of CVE-2023-1370.

project will encounter a stack overflow problem. To identify the vulnerability in the project, developers must search for all potential entries of the vulnerable function and verify if users can create an input resembling *s*.

```
StringBuilder s = new StringBuilder();
for (int i = 0; i < 10000 ; i++) {
  s.append("{\"a\":");
}
s.append("1");
for (int i = 0; i < 10000 ; i++) {
  s.append("}");
}
JSONParser p = null;
p = new JSONParser(JSONParser.MODE_JSON_SIMPLE);
p.parse(s.toString());
```

Figure 2: Vulnerability exploit for CVE-2023-1370 in jfrog.

Our objective is to identify vulnerabilities associated with CVE-2023-1370 in an open-source project named *microservice-with-jwt-and-microprofile*. Within the project, there is a code snippet that invokes `parse` as follows: `parser.parse(content)`. Given the project with the exploit of CVE-2023-1370, Vesta generates some test cases to trigger the vulnerability as clients of the project. Figure 3 presents the test case generated for this vulnerability, with line 4 denoting the invocation of the entry function. Replacing the parameter with a specific value such as the aforementioned *s* will result in the execution of this test case throwing an uncaught exception due to a buffer overflow problem. For projects with existing tests, Vesta will try to generate an exploit based on the graph from the test to the vulnerable function by replacing a triggerable value into the test directly.

With the assistance of Vesta, developers can execute the generated test case to ascertain if users can construct an input that triggers a buffer overflow in the project. For instance, executing `test05` in Figure 3 will cause result in a denial-of-service in the project, which means developers need to update the dependency version. Our method prevents false alarms by using vulnerability triggers as reliable evidence. Furthermore, we offer developers trigger test cases to assist in vulnerability identification in their projects and gain insight into how attackers may exploit the vulnerability.

```
@Test(timeout = 4000)
public void test05()  throws Throwable  {
  try {
    TokenUtil.of(POCValue);
    fail("Expecting exception: ClassCastException");
  } catch(ClassCastException e) {
     verifyException("TokenUtil",e);
  }
}
```

Figure 3: The generated test case for exploiting CVE-2023-1370 in the project named Microservice.

3 PROPOSED APPROACH

To generate test cases for triggering library vulnerabilities in projects, we implement a method called Vesta, as shown in Figure 4. Giving the binary file of the project and the exploit code for the library vulnerability, Vesta initially analyzes line coverage goals associated with the vulnerable function and assesses if the project's existing tests contain a call graph to the target function. Subsequently, Vesta generates test cases using a genetic algorithm to achieve the highest possibility to call the vulnerable function. Vesta generates tests that differ from those generated by EvoSuite in that they include call graphs passing parameters to the vulnerable function. These test cases are executed with an instrumentor and call graphs containing the vulnerable function are collected. Lastly, Vesta integrates the exploit code into the call graphs and assesses if the vulnerability is reproduced when executing the modified test cases.

3.1 Pre-Processing

Vesta takes the target project, along with its existing tests, as input. The pre-processing step has two major objectives: 1) To collect search goals for the generation step, including identifying the class name where the entry function is located and the position of the vulnerable function in the project. 2) To generate dynamic call graphs of the project's tests. If the analysis is based on the existing tests of the project, we compile and execute the current tests to obtain the dynamic invocation chain. If the analysis is based on the generated tests, we perform instrumentation analysis using the runtime package of EvoSuite.

To identify the search goals, we utilize a static analysis tool called *javacg-static* from Github[3]. The entry function is determined as the public method at the top of the static call graph containing the vulnerable function. The call graphs are obtained using a depth-first algorithm, which analyzes the function call relations collected by static analysis.

As shown in Algorithm 1, We design an instrumentor to collect the dynamic call graph during the test execution. The instrumentor inserts a push operator (line 5) before each function call and a pop operator (line 9) after each function call when executing tests. When the target function is called, the dynamic call graph containing the target function is stored on the stack (line 7).

3.2 Test Generation

After pre-processing, if no existing test is suitable for exploit migration, Vesta will generate test cases for the project, which contains a

[3]https://github.com/gousiosg/java-callgraph

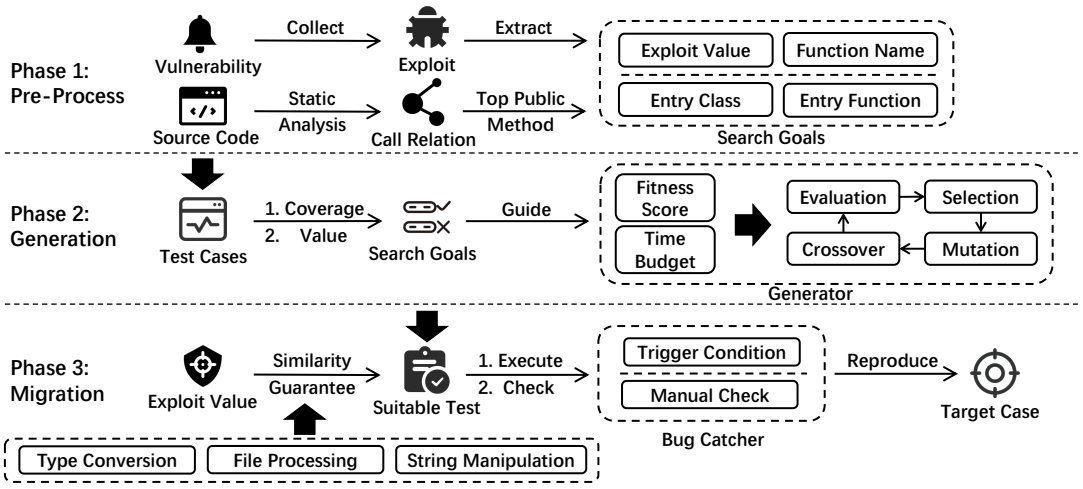

Figure 4: Overview of VESTA. Given the vulnerability exploits code, VESTA produces an exploit (Test Case) for the potential library vulnerability in the project.

Algorithm 1: Test Execution and Instrument

```
Input  : Test case files test_files, target function
         target_function
Output : Dynamic call graph call_graph
for test in test_files do
    if vulnerable_function in test then
        execute test
        for method in test do
            insertPushStatement(method)
            if method equals target_function then
                call_graph ← getCallGraph()
            end
            execute(method)
            insertPopStatement
        end
    end
end
```

call graph from the entry to the vulnerable function. Due to the maturity of EvoSuite [4, 16], VESTA reuses EvoSuite's genetic algorithm implementation.

Search Goals. In the migration step 3.3, VESTA selects a test case covering the target function and modifies the test to create an input that triggers the vulnerability in the vulnerable function. To achieve this objective, our search goals consist of two parts: 1) Ensuring that the generated test covers the vulnerable function. 2) Allowing the input value passed to the vulnerable function to be modified by adjusting the value provided in the generated test.

Generating a test case that invokes the vulnerable function can be challenging. To generate high coverage test cases for vulnerable functions in the project, VESTA collects three pieces of information as search goals: 1) *Entry Class Name* guides EvoSuite in generating tests for the function in the entry class, which represents how the client will call the function; 2) *Entry Function Name* is used to guide EvoSuite generating tests that contain the user entry function in the project, avoiding directly call the vulnerable function; 3) *Vulnerable Function Name* is used to verify whether the generated test can reach the vulnerable function. *Entry Class Name* and *Entry Function Name* are collected by our pre-processing step while *Vulnerable Function Name* is manually collected from the publicly disclosed vulnerabilities databases, like CVE [1].

To ensure that the generated test can pass values to the target function, we establish a search objective linked to the parameter value transmitted to the vulnerable function. We compare this value with the value in the exploit code, and if it changes during the test generation step, we can confirm the existence of a call graph that can provide crafted input to the vulnerable function. In some instances, this search goal leads to generating test cases that trigger the vulnerability directly.

Fitness Function. To generate tests that meet our search goals, we adjusted the fitness function in EvoSuite. As demonstrated in Algorithm 2, VESTA's fitness function assesses the proximity of the generated test to reach the library vulnerability function. The line coverage score measures the proximity of the test to accessing the vulnerable function, while the parameter similarity score evaluates the feasibility of passing the parameter from the test and triggering the vulnerability.

In VESTA, line coverage employs branch distance as a guiding metric for test generation. The branch distance helps generate tests that trigger vulnerable functions by measuring the proximity between the vulnerable function's branch and the actual branch taken by the generated test case.

To account for parameter similarity in the fitness score, we compute the similarity between the actual parameter value that the vulnerable function gets when executing the test and the value in the exploit based on the parameter types. Our work includes four types: number, string, file, and object.

Algorithm 2: Fitness Function for each Generation

```
Input  : The coverage goal goal, exploit value poc,
         generation gen
Output : Individual fitness score score
for ind in gen do
    (entry, function, value) ← execute(ind)
    score ← 0
    if entry in goal then
        score ← score + 1
    end
    if function in goal then
        score ← score + 1
    end
    switch value do
        case string do
            score ← score + distanceScore(value, poc)
        end
        case number do
            score ← score + isSameValue(value, poc)
        end
        case object do
            score ← score + inspectorScore(value, poc)
        end
        case file do
            obj ← convertToFileObject(value)
            score ← score + inspectorScore(obj, poc)
        end
    end
    return score
end
```

Number. In Java, the number type includes Int, Long, Short, Double, Float, char, and Boolean. Number type parameters are compared directly with the value in the exploit code. If the value in the generated test matches the value in the exploit, the similarity is 1, otherwise, the similarity is 0.

String. We use edit distance [37] to evaluate the similarity between the actual and exploit parameters, which represents the number of steps required to change one string to the target string. To calculate similarity, Vesta divides the edit distance by the length of the longer string and subtracts the result from 1. If the two strings are identical, the similarity is 1.

Object. Vesta calculates the similarity of two objects by the average similarity between inspectors [14, 27]. The corresponding similarity calculating methods are selected by the inspectors' type.

File. Files are treated as objects in Vesta due to their representation as File objects in Java. The exploit parameter is stored as a position on the disk. During the comparison, Vesta retrieves the exploit file from the disk and compares it with the actual value passed to the vulnerable function.

Implementation. We use javacg-static to get the location of the vulnerable function within the project's binary file. Our approach utilizes the existing infrastructure of EvoSuite 1.0.4 and involves modifying the fitness function within EvoSuite. The coverage of the vulnerable function is measured using branch distance. The similarity is assessed based on the parameter types of the vulnerable function, ensuring the presence of the call graph from the test to the vulnerable function. We use Javassist[4] to design our instrumentor, which collects dynamic call graphs during test execution and replaces entry parameter values while migrating.

3.3 Exploit Migration

Vesta includes a migration step in the generated test to trigger the vulnerability and significantly improves effectiveness by utilizing domain knowledge extracted from vulnerability exploits. Instead of using the triggering condition, we use the parameter value passed to the vulnerable function in the exploit to guide test generation. Additionally, rules are collected from the situations in which the parameters are modified in the process of passing value from the entry function to the vulnerable function.

Exploit Extraction. Triggering library vulnerabilities in a project is typically manifested as calls to vulnerable APIs. For example, Figure 5 illustrates the exploit for CVE-2021-44248, which invokes the `logger.error` function with a specific value. Executing the exploit triggers remote code execution caused by the Apache-log4j2[5] library. The vulnerability exploit contains information about the value passed to the vulnerable function, which is treated as domain knowledge for reproducing the library vulnerability. In Vesta, we collect the parameter value passed to the vulnerable function in the exploit by executing the exploit code snippets to ensure generate a test case that passes a similar value as the value in the exploit code.

```
1 private static final Logger logger = LogManager.
      getLogger(log4j.class);
2 public static void main(String[] args) {
3     logger.error("${jndi:ldap://localhost:8080/exploit}
          ");
4 }
```

Figure 5: The exploit code of CVE-2021-44248, will cause the exploit.class to run on the server (in the exploit, it is hosted on localhost).

Parameter Migration. Vesta utilizes the parameter value extracted from the exploit code to trigger the library vulnerability during the processing of the generated test cases. The fitness function of the genetic algorithm incorporates a calculation to assess the similarity between the test's passing value and the parameter value of the exploit. A test attains a high fitness score when a call graph that enables the parameter value to be passed from the test to the vulnerable function exists. Given the presence of this call graph, modifying the parameter at the entry of the call graph (Test) could alter the value received by the vulnerable function at the end of the call graph.

Instrumentor is a crucial component in Vesta which enables dynamic analysis and monitoring of program execution. It collects runtime information, such as dynamic call graphs, and allows for the modification of parameter values during execution.

[4]https://www.javassist.org
[5]https://logging.apache.org/log4j/2.x/

Table 1: Trigger conditions of vulnerabilities in VESTA.

Type	Show
DOS	Uncatch exception / Infinite loop
RCE	Target server receive request
Wrong Behavior	No exception throwed
SQL Injection	Database unexpected logs
XXE	Target server receive request

For a specific vulnerability, Vesta retrieves triggering parameter values from the exploit. By executing the generated test cases, we can obtain the entry function during testing. Vesta then re-executes the test with the instrumentor, which replaces the entry function's parameter value and records the target function's received value. When the executing entry function, its parameter will be replaced with the retrieved value and the parameter passed to the vulnerable function is recorded. If the entry function has more than one parameter, Vesta will traverse all positions and obtain the correct position to pass the value to the target function.

Trigger Capture. For each vulnerability type, Vesta defines the trigger condition of the vulnerability. Our experiment part includes these types: Denial of Service (DOS), Remote Code Execution (RCE), Function Wrong Behavior, SQL Injection, and XML External Entity Injection (XXE). Table 1 shows the definition of the conditions in Vesta. During the test execution process, Vesta collects the test execution conditions and checks if they meet our defined criteria. A test that matches the defined criteria is considered an exploit test for the library vulnerability in the project.

However, manual confirmation is still necessary to verify some vulnerabilities for the following reasons: (1) After fixing the vulnerability, the only observable change may be in the API return value without any other behaviors such as exceptions; (2) EvoSuite's sandbox implementation limits file access, so developers should log the return value of the vulnerable function or manually execute the generated test to check for trigger conditions.

Similarity Guarantee. Under certain conditions, solely replacing the value of the entry function may not trigger the vulnerability due to modifications within the call graph parameters. To overcome this issue, it is necessary to assess the potential for reproducing the vulnerability and modify our primitive value accordingly. By executing the migrated test and monitoring code conditions, Vesta assesses the possibility of the test triggering the vulnerability. If the test fails to reproduce the library vulnerability, Vesta utilizes collected rules to process the migrating parameter to ensure the similarity between generated test and the exploit.

As shown in Algorithm 3, during the execution of the test process, if the test fails to directly trigger the vulnerability, Vesta attempts to utilize rules to manipulate the parameter values within the test's entry function and re-execute the test, aiming to trigger the vulnerability. Typical rules encompass parameter type conversion, string manipulation, and file processing.

Parameter Type Conversion. The parameter type is usually converted from the input to the vulnerable function during the parameter transfer process. For example, the target function expects a parameter of type `ByteArray`, whereas the entry function requires a parameter of type `String`. During the migration process, Vesta obtains the parameter type from the exploit (Usually `String`), as well as the expected type of the entry function from the static analysis results. It then modifies the input type accordingly.

String Manipulation. String manipulation refers to the process of altering a String-like exploit value by appending a substring and incorporating it into an object or another format. Adding a substring involves inserting a string into specific locations within the value or incorporating the value into a predefined template string (e.g., inserting the server IP into an exploit string in RCE vulnerabilities). In some cases, the input value may represent only a portion of a specific format, such as a value within an object type. In such situations, the exploit string is inserted into an incomplete object or JSON template.

File Processing. Certain vulnerability exploits involve specific files that trigger vulnerabilities, and these files are stored at specific disk locations. In genetic algorithms, generating files with such specific characteristics can be challenging, even with knowledge of the file processing rules. To reduce the generation number and improve the correct rate, rather than collecting rules for file processing, we streamline the migration process by directly converting file positions into corresponding file objects in the executing process. Vesta accesses and converts these files into file objects during migration to facilitate subsequent processing.

Algorithm 3: Modifying Parameter Value

```
Input  : Generated test generated_test
Output : Modified test modified_test
modified_test ← generated_test;
if generated_test not in trigger_vulnerability then
    if parameter_type not equals trigger_type then
        generated_test ←
          typeConvertion(generated_test);
    end
    generated_test ← stringManipulation;
    reRexecution(generated_test);
end
return generated_test;
```

Exploit Generation. Following parameter migration, Vesta re-executes the migrated test. If the vulnerability trigger condition is satisfied, the test is reported as an exploit for the library vulnerability in the project. To better understand the vulnerability, the call graph from the entry function to the vulnerable function is provided. For projects without library vulnerability, Vesta generates tests equally but the generated tests are unable to reproduce the vulnerability.

Additionally, during the static analysis process (pre-process in Section 3.1), if the project's existing tests contain a call graph to take into the vulnerable function, Vesta will skip the test generation step and solely migrate the exploit parameter to the test. If the existing test fails to identify a library vulnerability, we will generate tests for the project as projects without satisfying tests.

Table 2: Vulnerabilities in our experiment, including vulnerability numbers, library names, vulnerability trigger conditions. Each Vulnerability has two projects, we indicate ✓ if the method is a success in the project, ✗ if the method failed in the project, —— means no vulnerability witness test is added after fixed or the fixing test is not merged.

Type	Library	Number	Function	Trigger Condition	VESTA	TRANSFER
Xml	XStream	CVE-2017-7957	fromXML	Uncatch Exception	✓✓	✓✓
		CVE-2021-39144	fromXML	Remote Code Execution	✓✓	——
		CVE-2021-21341	fromXML	Infinted Loop	✓✓	——
		CVE-2022-41966	fromXML	Stack Overflow	✓✓	✓✓
		CVE-2020-26217	fromXML	Remote Code Execution	✓✓	✗✗
Base64/32	Apache Codec	CODEC-263	decodeBase64	Wrong Behavior	✓✗	——
		CODEC-270	decodeBase64	Wrong Behavior	✓✓	✓✓
String	Apache Text	TEXT-215	translate	Wrong Behavior	✓✓	——
		CVE-2022-42889	replace	Remote Code Execution	✓✗	✗✗
Number	Apache Lang	LANG-1484	isParsable	Wrong Behavior	✓✗	——
		LANG-1645	createNumber	Wrong Behavior	✓✓	✗✗
		LANG-1385	createNumber	Wrong Behavior	✓✓	✗✗
Json	Json-smart	CVE-2023-1370	parse	Stack Overflow	✓✗	✗✗
		CVE-2021-27568	getAsNumber	Uncatch Exception	✓✗	✓✗
	JSON	CVE-2022-45688	parse	Uncatch Exception	✓✓	✓✗
	Jackson-databind	CVE-2019-14540	readValue	Remote Code Execution	✗✗	✗✗
File	Apache.poi	CVE-2019-12415	XSSFExportToXml	XXE Injection	✓✓	——
	Zip4j	CVE-2022-24615	ZipInputStream	Uncatch Exception	✓✗	——
		Zip-263	ZipFile	Wrong Behavior	✓✓	✓✓
	Apache IO	IO-611	normalize	Path Traversal	✓✓	✗✗
		CVE-2021-29425	normalize	Path Traversal	✓✓	✗✗
	Apache PDFBox	CVE-2021-31812	load	Stack Overflow	✓✓	——
Compress	Apache Compress	CVE-2021-35516	SevenZFile	Out of Memory	✓✗	✓✗
		CVE-2018-1324	ZipFile	Wrong behavior	✓✓	——
Test	Junit	CVE-2020-15250	TemporaryFolder	Improper File Permission	✗✗	✗✗
Framework	Spring-beans	CVE-2022-22965	ClassLoader	Remote Code Execution	✗✗	——
Net	Httpclient	CVE-2020-13956	HttpGet	Cross-site Scripting	✓✓	✗✗
HTML	Jsoup	CVE-2021-37714	parse	Infinted Loop	✓✗	✗✗
Log	Log4j2	CVE-2021-44228	error/info	Remote Code Execution	✓✗	✗✗
Database	Hibernate	CVE-2019-14900	getResultList	SQL injection	✗✗	✗✗
	18 Libraries	30 Vulnerabilities			43/60	11/60

4 EXPERIMENTS

We conduct an empirical evaluation to assess the effectiveness of our method in detecting library vulnerabilities. The evaluation aims to answer the following research questions:

- **RQ1. Is VESTA effective in generating exploits for library vulnerabilities?**
  This question focuses on assessing the effectiveness of Vesta in vulnerability discovery. We evaluate the accuracy of Vesta in a manually selected vulnerability dataset. Transfer [27] is used as the baseline for comparison to determine if both approaches can identify exploitable vulnerabilities and assess the time cost for their discovery.
- **RQ2. Does exploit migration enhance the effectiveness of VESTA?**
  The experimental section will discuss the performance of vulnerability detection based on migration compared to direct test generation, aiming to validate the advantages of exploit generation with our migration step.

4.1 Experimental Setup

Here, we discuss the experimental subjects used in our study, including the collected dataset and our baseline.

Context Selection. The experimental section analyzes 30 reported vulnerabilities from the past 5 years. Two projects associated with the corresponding library and affected by the vulnerabilities

Table 3: Vulnerable function parameter types in experiment.

Parameter Type	Vulnerability
String	20
File	3
Object	5
Number	2

are selected for experimentation for each vulnerability. We exclude toy projects from our dataset and only select those with 1000 or more code lines. We have a total of 41 projects, and we provide the complete list of projects on our website. To ensure the generalizability of the method, the dataset includes various types of vulnerability types, such as Denial of Service (Infinite Loops, Uncaught Exceptions), Wrong Function Results, Remote Code Execution, XML Data Injection, and SQL Injection. Additionally, experiments are conducted on diverse types of libraries with different functionalities, such as JSON processing, Java testing frameworks, Base64 conversion, and HTTP frameworks. Under this criterion, the vulnerabilities involved in the experiments are not limited to a single domain or a specific type. As Table 3, our experiment includes four common vulnerable parameter types, which manifests that our method is effective for different API-level vulnerabilities. We categorize the various conditions of vulnerability triggers into five types, which aim at avoiding manual checking of triggers. These types cover 8 of the 10 most common CWE in 2022 [33], with CSRF and XSS being the only exceptions.

For each vulnerability, the dataset includes two projects that depended on libraries affected by that vulnerability and one project did not affected by the vulnerability although exists the vulnerable function call. In order to compile these projects successfully and obtain more experimental data, we make certain modifications, such as changing dependency versions and removing uncompilable files, while avoiding any modifications to functions present in the vulnerable function call graphs to prevent false positives. Exploit code is selected from the report of each vulnerability, and the corresponding exploit parameters are extracted. Projects involved in experiments are Java projects managed by Maven.

To verify the feasibility of vulnerability exploit migration based on existing tests, we select four vulnerabilities and each two corresponding projects with comprehensive tests during project collection. We conduct experiments on the aforementioned vulnerabilities to validate the rationale of utilizing existing tests for discovering exploitable vulnerabilities. Moreover, the experimental section also designs a comparison between vulnerability discovery based on test generation and based on existing tests on the same projects to discuss the similarity between generated tests and the manually designed test. The vulnerabilities involved in the experiments are presented in Table 2.

During the experiment setup, vulnerability information is obtained from CVE or the libraries' vulnerability reports. The exploit code used in the experiment is sourced from various open-source forums (e.g., snyk [40]), and the vulnerable projects are sourced from GitHub, ensuring the authenticity of the experiment.

Baseline. We compare our method with Transfer [27], which generates tests for exploiting library vulnerabilities guided by the code behavior captured while executing vulnerability-witnessing tests. To run Transfer in our experiment, we collect positions of vulnerable functions within the projects and vulnerability-witness tests from vulnerable libraries. We evaluate Transfer in our experiments based on the example provided in the code package of Transfer.

Setup. To determine if a test triggers the vulnerability, Vesta captures the execution result of the generated tests and checks if a triggering behavior, corresponding to our defined trigger condition, has occurred. A test that exhibits the triggering behavior during execution is considered an exploit.

A test case is classified as an exploit if a defined trigger condition occurs and is detected by Vesta during test execution. Both methods will be executed 10 times for each project, and the method is considered to be effective if generates the exploit 5 times or more. However, if the method falsely determines a project without exploitable vulnerabilities as positive, it will be marked as a false positive.

We conduct our experiments on a 3.5GHz M2 device with 16GB RAM. Following Kang et al.'s experimental result [27], we set the test generation time budget in both approaches for 60 seconds.

4.2 Results

Here, we answer our two research questions by analyzing the experiment results.

4.2.1 RQ1: Effectiveness of Vesta. Table 2 presents the results of Vesta's detection of vulnerabilities. In projects containing exploitable vulnerabilities, Vesta identified 71.7% (43/60) of exploitable vulnerabilities while on the same dataset, Transfer only confirmed 18.3% (11/60) of exploitable vulnerabilities. Vesta outperformed Transfer on 20 vulnerabilities. This result demonstrates the effectiveness of Vesta in identifying exploitable library vulnerabilities and generating corresponding exploit code.

Our experiment evaluates the average time taken by Vesta for generating exploitable vulnerabilities. Based on existing tests, Vesta only requires preprocessing to obtain potential vulnerability call graphs and performs the migration step to generate an exploit. For projects without existing tests, the time search budget for the generated work during test generation is uniformly set to 10 seconds. In the experiments, all successful cases completed the test generation step within 10 seconds, demonstrating its superior performance in practice. On projects with tests containing exploitable call graphs, Vesta takes an average of 3.09 seconds to discover exploitable vulnerabilities. On projects without complete test coverage, the average time for discovering exploitable vulnerabilities is 22.75 seconds.

We run Vesta on projects with exploitable vulnerability call graphs that are unable to trigger vulnerabilities, causing false alarms in traditional vulnerability discovery methods. These projects are often mistakenly identified as having exploitable vulnerabilities in traditional vulnerability discovery methods like dependency analysis, leading to false alarms. In all 30 projects, Vesta consistently determines the absence of exploitable vulnerabilities, which aligns with the expected outcome.

Table 4: Results of Ablation Study on VESTA.

Method	Exploit	Effectiveness
Migration with Rules	11	18.3%
Directly Migration	27	45.0%
Directly Generated Test	5	8.3%
Total	43	71.6%

Table 5: VESTA's performance on projects with complete tests in three scenarios: direct migration, migration after generation, and test generation only.

Method	Found	Omitted	accuracy
Migration on Generated Test	8	0	100.0%
Migration on Existing Test	8	0	100.0%
Directly Generated Test	4	4	50.0%

Answer to RQ1. Vesta can generate exploit code for 43 projects associated with 26 third-party vulnerabilities, whereas the baseline Transfer can only generate exploit code for 11 projects. Furthermore, Vesta does not produce any false alarms, thereby demonstrating its reliability in the task of project vulnerability discovery.

4.2.2 RQ2: Ablation Study. We conduct an ablation study on Vesta by removing components from it. The results of this experiment are shown in Table 4. Compared to EvoSuite, Vesta incorporates vulnerability-driven migration tasks. Without migration tasks, Vesta can only generate 5 exploit tests related to 3 vulnerabilities (CODEC-263, CODEC-270, Zip-263), which aligns its performance with EvoSuite configured with line coverage and branch coverage. After incorporating the migration step, Vesta can generate exploits for an additional 38 projects, demonstrating the effectiveness of Vesta's vulnerability discovery based on migration.

Table 5 presents the ablation study conducted on projects with complete tests. Experiments are conducted on these projects using the following methods: 1. Migration using existing tests, 2. Migration using generated tests, and 3. Experimentation using only generated tests. This experiment demonstrates that migration based on existing tests and migration based on generated tests both exhibit good performance, while direct generation performs poorly.

Table 6 compares the time required for generating exploit code on the vulnerabilities in which both Vesta and Transfer have successful cases. Since Transfer relies on manual confirmation for locating the vulnerable function positions in the projects, the experiment omits the time spent on this part and focuses only on comparing the generation time in Transfer with the generation and migration time in Vesta. This experiment revealed that when complete tests already exist for the projects, Vesta can rapidly generate exploit code based on the existing project tests. Furthermore, the time performance for migration based on generated tests is also better than the direct generation.

Answer to RQ2. Vesta's migration part achieves a success rate of 63.3% and showed an average improvement of 7.81 seconds in time compared to Transfer. Therefore, the approach of migration based on test generation exhibits efficiency and reliability in discovering third-party exploitable vulnerabilities in projects.

Table 6: Exploit Generation Time. In VESTA, we evaluate time both using existing tests and generating tests.

Vulnerability	VESTA	TRANSFER
CVE-2017-7957	2.20s/18.71s	31s
CVE-2022-41966	2.51s/18.41s	32s
CODEC-270	3.51s/16.33s	12s

5 DISCUSSION

In this part, we discuss the reliability and threats to validity of our study.

5.1 Qualitative Analysis

The fitness function of Transfer in test generation aims to ensure the generated tests exhibit similar code behavior as the tests added to the vulnerable repository after fixed. However, this approach presents several issues. Firstly, some vulnerabilities, such as CVE-2022-24615, do not have additional tests added after the vulnerability is repaired (11 vulnerabilities). Meanwhile, a no runnable methods exception will occur if the libraries' test framework is JUnit5 instead of JUnit4 (8 vulnerabilities). Another problem arises when the vulnerability repository tests invoke methods specific to the test classes, which cannot be called in the target project, as exemplified by CVE-2020-26217. Lastly, tests generated based on code behavior still exhibit differences compared to actual tests capable of triggering vulnerabilities, particularly when file operations are involved, as demonstrated by CVE-2019-12415.

In contrast, Vesta exploits vulnerability by generating tests and obtaining exploitable call graphs, ensuring that all methods called in the tests are existing methods within the project. In the experiment involving CVE-2020-26217, Transfer utilized a code remote execution detection method from the XStream library test. However, since developers require to execute the test within their projects, the relevant files for this method are not included. As a result, Transfer encountered a `CannotResolveClassException` error, indicating the inability to locate the aforementioned files, thereby failing to discover exploitable vulnerabilities. In contrast, Vesta employs the most primitive remote execution method in vulnerability exploitation, directly accessing `localhost` to ensure the reliable triggering of vulnerabilities. Additionally, in CVE-2020-15250, Transfer created a test that invoked a private function in JUnit. However, due to the method's inaccessibility in the server project, this test could not evaluate the exploitability of the vulnerabilities.

Moreover, the utilization of migration ensures the stability of vulnerability triggering and proves more effective than the code behavior-guided approach employed by Transfer. This migration strategy guarantees Vesta's performance in complex scenarios. For example, in CVE-2019-12415, reproducing the vulnerability requires passing a specially crafted .xlsx file with a remote execution address included in the file header. Generating such a file through genetic algorithms poses significant challenges, resulting in failures

for EvoSuite. However, VESTA directly triggers the vulnerability by passing a pre-built exploit file, demonstrating its ability to reliably exploit the vulnerability. In complex scenarios, the migration strategy effectively ensures the reliability of vulnerability triggering.

After analyzing the above conclusions, VESTA ensures the stability of reproducing test vulnerabilities through a migration-based strategy. This is particularly crucial when dealing with complex scenarios involving parameters such as files. Additionally, VESTA improves the performance of generating tests on the modified call chains by applying rules to modify the passed parameters during the transmission process. Defining the manifestation of vulnerability triggers reduces the need for manual checks and minimizes the impact on the methods.

5.2 Threats to Validity

External Validity. One possible factor that may affect the authenticity of the experiment is that the dataset is manually selected, raising concerns about the coverage of vulnerability domains and exploit types. Additionally, We limit our selection to projects with reachable exploit code, which may also be a limitation. Some trigger conditions are not covered in our method, which means a manual check is also required.

Internal Validity. In certain instances, VESTA encounters difficulties in generating suitable tests due to the following reasons: 1) Project-related issues, such as the inability to analyze jar files, failures in loading classes in EvoSuite, and uncompilable projects, result in 5 failure cases; 2) the generated tests failed in covering vulnerable function. For example, there is a failure in vulnerability CODEC-263 within the project named java-algorand-sdk. In this case, an if-branch checks one of the input values' types, which should be a valid image format, before reaching the vulnerable function. However, our generated test passes an illegal input, thus preventing the reaching of the vulnerable function.

Despite the aforementioned limitations, our method produces significantly better results than our baseline in a dataset with sufficient types and quantities of vulnerabilities. We are confident that our method has considerable generalizability.

6 RELATED WORK

Software Composition Analysis. Software composition analysis is a domain that involves managing vulnerable dependencies in software projects, which includes identifying, tracking, and resolving such vulnerabilities [15]. Prior researchers had cross-referenced the dependency versions used in a project against a database of known vulnerable versions to identify potential library vulnerabilities [40, 41]. Methods for dependency analysis would generate false positives [2, 15, 19], as less than 1% of packages have a reachable call path to vulnerable code [34]. Certain static analysis methods [36, 38] that rely on generating a static call graph of a software project and identifying potentially vulnerable functions might produce false positives due to discrepancies between the static call graph and the actual run-time behavior of the program [26]. They did not detect whether a vulnerability is reachable, which means whether an attacker can generate an input that passes to the vulnerable function and triggers the dependency vulnerability [21, 39]. Dynamic methods involved running the test cases of a software project to generate a call graph and got the control flow. However, these methods are limited by differences between the test cases and the actual code that can trigger dependency vulnerabilities, as well as by limitations in the test coverage and the ability to trigger vulnerabilities under real conditions [29, 30].

Kang et al. noted that the previously mentioned methods didn't consider control flow and if client projects were able to construct an input to trigger the vulnerability [25, 27, 38]. SIEGE utilized the coverage of the vulnerable function as a search criterion to generate test cases that call the target function [26], providing evidence that the library vulnerability can be reached. To meet the requirement of domain knowledge, Kang et al. manually selected test cases added to the library after fixing vulnerabilities, which is related to reproducing the vulnerability. Executing these vulnerability-witnessing test cases helped obtain the performance when the vulnerability is triggered. This performance serves as the criterion for generating test cases that satisfy the triggering condition [27].

Different from the aforementioned methods, VESTA employs the exploit code of the library vulnerability, which collects from open-source forums, to guide test case generation for a client project, which contains more precise domain knowledge. Additionally, the trigger behavior of the vulnerability is reproducing the vulnerability rather than relying on code performance, providing a more persuasive vulnerability exploit test.

Test Case Generation. EvoSuite is a tool that generates test cases with assertions for classes written in Java code by genetic algorithm [22, 44]. JUnit tests are represented as individuals and fitness scores are optimized using mutation, crossover, and other operators. Kang et al. and Iannone et al. modified the fitness score in EvoSuite to generate tests that can trigger the library vulnerabilities [26, 27]. However, these methods lack domain knowledge, despite Kang et al. use vulnerability-witness tests in vulnerable libraries to guide test generation [26, 27].

In addition to search-based methods, deep learning-based methods are widely used in test case generation. VDiscover used static and dynamic features to predict if a test case is likely to trigger a software vulnerability using machine learning techniques [23]. VulDeePecker initiated the study of using deep learning-based vulnerability detection to relieve human experts from the tedious and subjective task of manually defining features, leading to the design and implementation of a deep learning-based vulnerability detection system [32].

In our study, we use target function coverage and the similarity between generated test cases and the exploit code to guide test generation and add an exploit code migration step to the test case generated by EvoSuite, which enables us to modify the entry function's parameters and ensures the vulnerability function to receive a value causing vulnerability triggering.

Vulnerability Exploit Generation. Exploit code [7] is commonly used to detect vulnerabilities and implement defensive measures by exploiting software vulnerabilities during execution, such as taking control of computer systems, causing buffer overflows, or executing unexpected code [5]. Over the past decade, a significant amount of research has focused on the generation of exploit code for software projects [6, 17, 47].

AEG generated exploit code using binary information during code execution, but this approach was not universally applicable [6]. Xu et al. used symbolic execution to search the target software and found potential buffer overflow vulnerabilities, then generated the exploit of software vulnerability [47]. However, it did not perform well in complicated programs. AngErza used dynamic and symbolic execution to identify hot spots in the code, formulate constraints and generate a payload based on those constraints [17].

Our work relies on exploit code for the vulnerable library functions to guide test generation. We manually select exploit code from open-source websites due to the diversity of vulnerabilities and the requirement for a clear exploit code.

7 CONCLUSION AND FUTURE WORK

In this paper, we propose a method Vesta which utilizes the genetic algorithm to generate test cases covering vulnerable functions in the project. By migrating the vulnerability exploit code into the generated test, we construct an exploit test case for the library vulnerability in the project. Compared to Transfer, our unique migration step results in generating 53.4% more exploits. Executing the test case provides a reference for the developer to check if the project has exploitable library vulnerability and decide whether to update the dependency version. In the experimental evaluation, we test our method in 60 vulnerability-project pairs and receive 43 exploits, which shows its efficacy.

In the future, we plan to perform more experiments with more datasets to further evaluate the performance of our method. We will also explore the use of Large-scale Language Models to find library vulnerabilities.

ACKNOWLEDGEMENT

This research was supported by the National Natural Science Foundation of China (No. 62141222) and the National Research Foundation, Singapore under its Industry Alignment Fund – Prepositioning (IAF-PP) Funding Initiative. 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.

REFERENCES

[1] [n. d.]. NATIONAL VULNERABILITY DATABASE. https://nvd.nist.gov/vuln
[2] Mahmoud Alfadel, Diego Elias Costa, and Emad Shihab. 2021. Empirical Analysis of Security Vulnerabilities in Python Packages. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 446–457. https://doi.org/10.1109/SANER50967.2021.00048
[3] Sameed Ali, Prashant Anantharaman, and Sean W. Smith. 2020. Armor Within: Defending Against Vulnerabilities in Third-Party Libraries. In *2020 IEEE Security and Privacy Workshops (SPW)*. 291–299. https://doi.org/10.1109/SPW50608.2020.00063
[4] 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 *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. 263–272. https://doi.org/10.1109/ICSE-SEIP.2017.27
[5] I. Arce. 2004. The shellcode generation. *IEEE Security & Privacy* 2, 5 (2004), 72–76. https://doi.org/10.1109/MSP.2004.87
[6] Thanassis Avgerinos, Sang Kil Cha, Alexandre Rebert, Edward J. Schwartz, Maverick Woo, and David Brumley. 2014. Automatic Exploit Generation. *Commun. ACM* 57, 2 (feb 2014), 74–84. https://doi.org/10.1145/2560217.2560219
[7] Tiffany Bao, Ruoyu Wang, Yan Shoshitaishvili, and David Brumley. 2017. Your Exploit is Mine: Automatic Shellcode Transplant for Remote Exploits. In *2017 IEEE Symposium on Security and Privacy (SP)*. 824–839. https://doi.org/10.1109/SP.2017.67
[8] Gabriele Bavota, Gerardo Canfora, Massimiliano Di Penta, Rocco Oliveto, and Sebastiano Panichella. 2015. How the apache community upgrades dependencies: an evolutionary study. *Empirical Software Engineering* 20 (2015), 1275–1317.
[9] Christopher Bogart, Christian Kästner, James Herbsleb, and Ferdian Thung. 2016. How to break an API: cost negotiation and community values in three software ecosystems. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 109–120.
[10] Mircea Cadariu, Eric Bouwers, Joost Visser, and Arie van Deursen. 2015. Tracking known security vulnerabilities in proprietary software systems. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. 516–519. https://doi.org/10.1109/SANER.2015.7081868
[11] Sen Chen, Lingling Fan, Chunyang Chen, Minhui Xue, Yang Liu, and Lihua Xu. 2021. GUI-Squatting Attack: Automated Generation of Android Phishing Apps. *IEEE Transactions on Dependable and Secure Computing* 18, 6 (2021), 2551–2568. https://doi.org/10.1109/TDSC.2019.2956035
[12] Yang Chen, Andrew E. Santosa, Asankhaya Sharma, and David Lo. 2020. Automated Identification of Libraries from Vulnerability Data. In *2020 IEEE/ACM 42nd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. 90–99.
[13] Alexander Cobleigh, Martin Hell, Linus Karlsson, Oscar Reimer, Jonathan Sönnerup, and Daniel Wisenhoff. 2018. Identifying, Prioritizing and Evaluating Vulnerabilities in Third Party Code. In *2018 IEEE 22nd International Enterprise Distributed Object Computing Workshop (EDOCW)*. 208–211. https://doi.org/10.1109/EDOCW.2018.00038
[14] 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* (Shanghai, China) *(WODA '06)*. Association for Computing Machinery, New York, NY, USA, 17–24. https://doi.org/10.1145/1138912.1138918
[15] Alexandre Decan, Tom Mens, and Eleni Constantinou. 2018. On the Impact of Security Vulnerabilities in the Npm Package Dependency Network. In *Proceedings of the 15th International Conference on Mining Software Repositories* (Gothenburg, Sweden) *(MSR '18)*. Association for Computing Machinery, New York, NY, USA, 181–191. https://doi.org/10.1145/3196398.3196401
[16] Xavier Devroey, Sebastiano Panichella, and Alessio Gambi. 2020. Java Unit Testing Tool Competition: Eighth Round. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops* (Seoul, Republic of Korea) *(ICSEW'20)*. Association for Computing Machinery, New York, NY, USA, 545–548. https://doi.org/10.1145/3387940.3392265
[17] Shruti Dixit, T K Geethna, Swaminathan Jayaraman, and Vipin Pavithran. 2021. AngErza: Automated Exploit Generation. In *2021 12th International Conference on Computing Communication and Networking Technologies (ICCCNT)*. 1–6. https://doi.org/10.1109/ICCCNT51525.2021.9579959
[18] Johannes Düsing and Ben Hermann. 2022. Analyzing the Direct and Transitive Impact of Vulnerabilities onto Different Artifact Repositories. *Digital Threats* 3, 4, Article 38 (feb 2022), 25 pages. https://doi.org/10.1145/3472811
[19] Rodrigo Elizalde Zapata, Raula Gaikovina Kula, Bodin Chinthanet, Takashi Ishio, Kenichi Matsumoto, and Akinori Ihara. 2018. Towards Smoother Library Migrations: A Look at Vulnerable Dependency Migrations at Function Level for npm JavaScript Packages. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 559–563. https://doi.org/10.1109/ICSME.2018.00067
[20] FasterXML. [n. d.]. FasterXML/jackson-databind. https://github.com/FasterXML/jackson-databind
[21] Darius Foo, Jason Yeo, Hao Xiao, and Asankhaya Sharma. 2019. The Dynamics of Software Composition Analysis. *CoRR* abs/1909.00973 (2019). arXiv:1909.00973 http://arxiv.org/abs/1909.00973
[22] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: Automatic Test Suite Generation for Object-Oriented Software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering* (Szeged, Hungary) *(ESEC/FSE '11)*. Association for Computing Machinery, New York, NY, USA, 416–419. https://doi.org/10.1145/2025113.2025179
[23] Gustavo Grieco, Guillermo Luis Grinblat, Lucas Uzal, Sanjay Rawat, Josselin Feist, and Laurent Mounier. 2016. Toward Large-Scale Vulnerability Discovery Using Machine Learning. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy* (New Orleans, Louisiana, USA) *(CODASPY '16)*. Association for Computing Machinery, New York, NY, USA, 85–96. https://doi.org/10.1145/2857705.2857720
[24] Stefanus A. Haryono, Hong Jin Kang, Abhishek Sharma, Asankhaya Sharma, Andrew Santosa, Ang Ming Yi, and David Lo. 2022. Automated Identification of Libraries from Vulnerability Data: Can We Do Better?. In *2022 IEEE/ACM 30th International Conference on Program Comprehension (ICPC)*. 178–189. https://doi.org/10.1145/3524610.3527893
[25] Hong Hu, Zheng Leong Chua, Sendroiu Adrian, Prateek Saxena, and Zhenkai Liang. 2015. Automatic Generation of Data-Oriented Exploits. In *24th USENIX Security Symposium (USENIX Security 15)*. USENIX Association, Washington, D.C., 177–192. https://www.usenix.org/conference/usenixsecurity15/technical-

sessions/presentation/hu
[26] Emanuele Iannone, Dario Di Nucci, Antonino Sabetta, and Andrea De Lucia. 2021. Toward Automated Exploit Generation for Known Vulnerabilities in Open-Source Libraries. In *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*. 396–400. https://doi.org/10.1109/ICPC52881.2021.00046
[27] 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* (Virtual, South Korea) *(ISSTA 2022)*. Association for Computing Machinery, New York, NY, USA, 276–288. https://doi.org/10.1145/3533767.3534398
[28] Miryung Kim, Dongxiang Cai, and Sunghun Kim. 2011. An empirical investigation into the role of API-level refactorings during software evolution. In *Proceedings of the 33rd international conference on software engineering*. 151–160.
[29] 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 17th European Conference on Software Maintenance and Reengineering*. 353–356. https://doi.org/10.1109/CSMR.2013.48
[30] Pavneet Singh Kochhar, Ferdian Thung, and David Lo. 2015. Code coverage and test suite effectiveness: Empirical study with real bugs in large systems. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. 560–564. https://doi.org/10.1109/SANER.2015.7081877
[31] Raula Gaikovina Kula, Daniel M German, Ali Ouni, Takashi Ishio, and Katsuro Inoue. 2018. Do developers update their library dependencies? An empirical study on the impact of security advisories on library migration. *Empirical Software Engineering* 23 (2018), 384–417.
[32] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. 2018. Vuldeepecker: A deep learning-based system for vulnerability detection. *arXiv preprint arXiv:1801.01681* (2018).
[33] Mend. [n. d.]. The State of Open Source Security Vulnerabilities Annual Report 2021. https://www.mend.io/wp-content/media/2021/03/The-state-of-open-source-vulnerabilities-2021-annual-report.pdf
[34] Amir M. Mir, Mehdi Keshani, and Sebastian Proksch. 2023. On the Effect of Transitivity and Granularity on Vulnerability Propagation in the Maven Ecosystem. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 201–211. https://doi.org/10.1109/SANER56733.2023.00028
[35] Samim Mirhosseini and Chris Parnin. 2017. Can automated pull requests encourage software developers to upgrade out-of-date dependencies?. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 84–94. https://doi.org/10.1109/ASE.2017.8115621
[36] Benjamin Barslev Nielsen, Martin Toldam Torp, and Anders Møller. 2021. Modular Call Graph Construction for Security Scanning of Node.Js Applications. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, Denmark) *(ISSTA 2021)*. Association for Computing Machinery, New York, NY, USA, 29–41. https://doi.org/10.1145/3460319.3464836
[37] Constituency Parsing. 2009. Speech and language processing. *Power Point Slides* (2009).
[38] 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 *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 449–460. https://doi.org/10.1109/ICSME.2018.00054
[39] Serena Elisa Ponta, Henrik Plate, and Antonino Sabetta. 2020. Detection, Assessment and Mitigation of Vulnerabilities in Open Source Dependencies. *Empirical Softw. Engg.* 25, 5 (sep 2020), 3175–3215. https://doi.org/10.1007/s10664-020-09830-x
[40] Snyk. [n. d.]. Snyk. https://snyk.io/
[41] W. Software. [n. d.]. Whitesource. https://www.whitesourcesoftware.com/open-source-security/
[42] Synopsys. [n. d.]. OPEN SOURCE SECURITY AND RISK ANALYSIS REPORT 2023. https://www.synopsys.com/software-integrity/resources/analyst-reports/open-source-security-risk-analysis.html#
[43] Chongbin Tang, Sen Chen, Lingling Fan, Lihua Xu, Yang Liu, Zhushou Tang, and Liang Dou. 2019. A Large-Scale Empirical Study on Industrial Fake Apps. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. 183–192. https://doi.org/10.1109/ICSE-SEIP.2019.00028
[44] Christian Von Lücken, Benjamín Barán, and Carlos Brizuela. 2014. A survey on multi-objective evolutionary algorithms for many-objective problems. *Computational optimization and applications* 58 (2014), 707–756.
[45] 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 *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 35–45. https://doi.org/10.1109/ICSME46990.2020.00014
[46] James Wetter. [n. d.]. Understanding the Impact of Apache Log4j Vulnerability. https://security.googleblog.com/2021/12/understanding-impact-of-apache-log4j.html
[47] Luhang Xu, Weixi Jia, Wei Dong, and Yongjun Li. 2018. Automatic Exploit Generation for Buffer Overflow Vulnerabilities. In *2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. 463–468. https://doi.org/10.1109/QRS-C.2018.00085
[48] Huijie Yuan, Yunchao Wang, Guoxiao Zong, and Zhuo Lv. 2022. Exploitability Analysis of Public Component Library Vulnerabilities Based on Taint Analysis. In *2022 7th International Conference on Intelligent Computing and Signal Processing (ICSP)*. 1066–1072. https://doi.org/10.1109/ICSP54964.2022.9778489
[49] Xian Zhan, Lingling Fan, Sen Chen, Feng We, Tianming Liu, Xiapu Luo, and Yang Liu. 2021. ATVHunter: Reliable Version Detection of Third-Party Libraries for Vulnerability Identification in Android Applications. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 1695–1707. https://doi.org/10.1109/ICSE43902.2021.00150