MAGNETO: A Step-Wise Approach to Exploit Vulnerabilities in Dependent Libraries via LLM-Empowered Directed Fuzzing

Zhuotong Zhou*
School of Computer Science
Fudan University
Shanghai, China

Yongzhuo Yang*
School of Computer Science
Fudan University
Shanghai, China

Susheng Wu*
School of Computer Science
Fudan University
Shanghai, China

Yiheng Huang*
School of Computer Science
Fudan University
Shanghai, China

Bihuan Chen*†
School of Computer Science
Fudan University
Shanghai, China

Xin Peng*
School of Computer Science
Fudan University
Shanghai, China

ABSTRACT

The wide adoption of open source third-party libraries can propagate vulnerabilities that originally exist in third-party libraries through dependency chains to downstream projects. To mitigate this security risk, vulnerability exploitation analysis has been proposed to further reduce false positives of vulnerability reachability analysis. However, existing approaches work less effectively when the vulnerable function of the vulnerable library is indirectly invoked by a client project through a call chain of multiple steps.

To address this problem, we propose a step-wise approach, named MAGNETO, to exploit vulnerabilities in dependent libraries of a client project through LLM-empowered directed fuzzing. Its core idea is to decompose the directed fuzzing for the whole call chain (from the client project to the vulnerable function) into a series of step-wise directed fuzzing for each step of the call chain. To empower directed fuzzing, it leverages LLM to facilitate the initial seed generation. Our evaluation has demonstrated the effectiveness of MAGNETO over the state-of-the-art; i.e., MAGNETO achieves an improvement of at least 75.6% in successfully exploiting the vulnerability.

CCS CONCEPTS

• **Security and privacy → Vulnerability management**.

KEYWORDS

library vulnerabilities, exploit generation, directed fuzzing

ACM Reference Format:
Zhuotong Zhou, Yongzhuo Yang, Susheng Wu, Yiheng Huang, Bihuan Chen, and Xin Peng. 2024. MAGNETO: A Step-Wise Approach to Exploit Vulnerabilities in Dependent Libraries via LLM-Empowered Directed Fuzzing. In *39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24), October 27-November 1, 2024, Sacramento, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3691620.3695531

*Z. Zhou, Y. Yang, S. Wu, Y. Huang, B. Chen, and X. Peng are also with Shanghai Key Laboratory of Data Science, Fudan University, China.
†B. Chen is the corresponding author.

ASE '24, October 27-November 1, 2024, Sacramento, CA, USA

ACM ISBN 979-8-4007-1248-7/24/10
https://doi.org/10.1145/3691620.3695531

1 INTRODUCTION

Open source third-party libraries play a crucial role in modern software development, providing a wealth of foundational functionalities. Developers can reuse these functionalities by declaring third-party libraries as dependencies, significantly reducing development time and labor cost [23, 27, 30, 39, 45]. Therefore, more and more developers adopt third-party libraries in their projects. Maven Central, as the most widely used library repository in the Java ecosystem, records an average of 37.8 billion downloads per month [38].

In fact, 3.97 of the 37.8 billion monthly downloads consume vulnerable libraries [38]. Therefore, this practice can propagate vulnerabilities that originally exist in third-party libraries through dependency chains to downstream projects, which exposes these projects to security threat of these vulnerabilities. For example, the Log4Shell vulnerability (CVE-2021-44228) in the Apache Log4j library affected 35,863 Java libraries as of December 16, 2021, amounting to over 8% of all libraries on Maven Central [20]. Besides, it is often difficult for downstream projects to timely remediate these vulnerabilities as vulnerable libraries can be transitively depended on. The deeper the vulnerability is in a dependency chain, the more steps are needed for it to be fixed. For example, only around 7,000 (19.5%) of the affected libraries of the Log4Shell vulnerability are direct dependencies [20]. Moreover, after the vulnerabilities are fixed by releasing a new library version, developers often neglect timely dependency upgrade, due to the potentially introduced compatibility issues [32, 45]. This leaves their projects exposed to the security risk of exploitation.

Problem. Various approaches have been proposed to analyze and mitigate the security risk of vulnerabilities in dependencies. In particular, *vulnerability existence analysis* employs software composition analysis to extract a project's dependency tree, and then checks the presence of vulnerable dependencies [22, 31, 56]. However, they do not consider whether the project invokes the vulnerable functions, and hence incur high false positives. To resolve this problem, *vulnerability reachability analysis* adopts call graph generation to build a project's call graph, and then checks the presence of call chains through which to reach the vulnerable functions [24, 33, 37, 50, 53, 57]. However, they fail to consider whether the project triggers the vulnerabilities, and thus still cause false positives.

To address this issue, *vulnerability exploitation analysis* [10, 25, 28] has been recently proposed. SIEGE [25] uses the vulnerable code in a vulnerable library as the goal, and guides EVOSUITE [14] to generate test cases for the project to reach the vulnerable code. Differently, TRANSFER [28] uses the program state when the vulnerable function is called in a vulnerable library's vulnerability-triggering test case as the goal, and guides EVOSUITE to generate test cases for the project to call the vulnerable function with the same program state. VESTA [10] uses EVOSUITE to generate test cases for the project to reach the vulnerable function, and then migrates parameters from a vulnerable library's vulnerability-triggering test case into these generated test cases based on predefined rules. These generated test cases by these approaches are considered as exploits to trigger the vulnerability in the dependent library from the project side.

These approaches work well in simple scenarios where the vulnerable function is directly invoked by the project (i.e., the call chain from the project to the vulnerable function is one-step). However, in complex scenarios where the vulnerable function is indirectly invoked by the project through a call chain of multiple steps, the exploitation process becomes intrinsically complex, making these approaches hard to generate an effective exploit. As reported by Wu et al.' work [50], the majority of call chains to the vulnerable function have multiple steps. Unfortunately, the majority of the experimental ground-truth data set in these approaches consider simple scenarios rather than complex scenarios, making their practical effectiveness in complex scenarios not comprehensively evaluated.

Our Approach. To overcome the limitation of existing approaches, we propose a step-wise approach, named MAGNETO, to exploit vulnerabilities in dependent libraries of a client project through LLM-empowered directed fuzzing. The core idea of MAGNETO is to break down the directed fuzzing for the whole call chain (from the client project to the vulnerable function) into a series of step-wise directed fuzzing for each step of the call chain in an reverse order. In other words, we decompose the whole difficult exploit generation task into a series of relatively simple exploit generation tasks.

Specifically, given a client project and a library's exploit (in the form of a test case) that triggers a vulnerability in the library, MAGNETO works in two steps. First, it performs vulnerability reachability analysis to identify all the call chains that statically reach the vulnerable function. Then, for each step (in the form of a pair $\langle F_i, F_{i-1} \rangle$ meaning that F_i calls F_{i-1}) of each of the call chains in an reverse order (i.e., F_{i-1} is the vulnerable function in the library in the first step), MAGNETO conducts LLM-empowered directed fuzzing to generate the exploit. During this step-wise directed fuzzing, MAGNETO generates an initial seed based on the LLM's semantic understanding of F_i as well as the exploit generated in the previous step (or the given exploit if in the first step). This initial seed is helpful to enhance the likelihood of successful exploit generation. Using this initial seed, MAGNETO runs heuristic-directed fuzzing to generate the exploit. MAGNETO can be adopted by developers to discover exploitable vulnerabilities in project dependencies, and provide them with reachable call chains and exploits as confident evidences to quickly mitigate the security risk in dependencies.

Evaluation. We evaluate MAGNETO and the state-of-the-art approaches SIEGE [25], TRANSFER [28] and VESTA [10]. We collect 32 vulnerabilities (affecting 21 third-party libraries and 45 vulnerable functions) and 49 GitHub projects which can exploit the vulnerabilities.

```
1 public void testSuppressClassPropertyByDefault() throws Exception {
2    Object bean = new Object();
3    try {
4      Object clazz = PropertyUtils.getProperty(bean, "class");
5      fail("Could access class property!"); // Exploit Success
6   } catch (final NoSuchMethodException ex) {
7      // Exploit Fail
8   }
9 }
```

Figure 1: An Exploit for CVE-2019-10086

In total, we collect 84 pairs of vulnerability and project (where the project can exploit the vulnerability) and 182 exploitable call chains. MAGNETO successfully generates an exploit for 79 pairs and 135 call chains, while the best of the state-of-the-art successfully generates an exploit for 45 pairs and 53 call chains. This indicates an improvement of at least 75.6% and 154.7% over the state-of-the-art. MAGNETO averagely takes 10.9 minutes to exploit a vulnerability for a project, which is acceptable considering its significant effectiveness.

Contribution. This work makes the following contributions.

- We proposed a step-wise approach, named MAGNETO, to generate exploits for vulnerabilities in dependent libraries.
- We conducted extensive experiments to demonstrate the effectiveness and efficiency of MAGNETO.

2 MOTIVATING EXAMPLE

Before version 1.9.3 of Apache commons-beanutils [15], there was a vulnerability identified as CVE-2019-10086. It allowed attackers to access the classloader via the class property available on all Java objects, leading to security risks [34]. Fig. 1 shows an exploit of this vulnerability. The exploit passes two parameters to call the vulnerable function `PropertyUtils.getProperty`, a regular object `bean` and a string `"class"` (Line 4). As `bean` does not have a class property, this function call is expected to throw a `NoSuchMethodException`. However, this function call returns the unexpected class object of `Object` (i.e., `Object.class`), thereby triggering the vulnerability.

Prior to version 1.7, commons-validator [16] depended on a vulnerable version of commons-beanutils that had CVE-2019-10086 [1]. As shown by Fig. 2, there exists a reachable call chain from a function in commons-validator to the vulnerable function in commons-beanutils, which is `Form.validate` → `Field.validate` → `Field.getIndexedPropertySize` → `PropertyUtils.getProperty`. We denote the functions along this call chain in the reverse order as F_0, F_1, F_2 and F_3. The existence of a reachable call chain does not mean that the vulnerability can be exploited. Therefore, we need to conduct exploitability analysis, which brings some challenges.

Challenge 1: How to evaluate the exploitability of a vulnerability through a reachable call chain?

In the last function call (i.e., F_1 calls F_0 at Line 59), the second argument of F_0 (i.e., the return value of `getIndexedListProperty`) is required to be `"class"` to exploit the vulnerability. This means that the value of the `indexedListProperty` field in the receiver object of F_1 should be `"class"`. When this receiver object executes F_1, passing a regular instance of `Object` can exploit the vulnerability in F_0. TRANSFER mimics the program state during the vulnerability-triggering test case execution (i.e., the test case in Fig. 1), and hence successfully generates an exploit for F_1. However, VESTA only considers function arguments directly passed to vulnerable functions,

```
1  class Form { // Form.java
2     protected List<Field> lFields;
3     protected Map<String, Field> hFields;
4
5     ValidatorResults validate(Map<String, Object> params, ...,
6                               int page, String fieldName) {
7     ValidatorResults results = new ValidatorResults();
8     if (fieldName != null) {
9      Field field = hFields.get(fieldName);
10       if (field == null) { throw exception }
11       if (field.getPage() <= page) {
12          field.validate(params, ...); /* 1st Invocation */
13       }
14    } else {
15       Iterator<Field> fields = this.lFields.iterator();
16       while (fields.hasNext()) {
17       Field field = fields.next();
18       if (field.getPage() <= page) {
19          field.validate(params, ...); /* 1st Invocation */
20        }
21       }
22     }
23     return results;
24    }
25 }
26
27 class Field { // Field.java
28   String depends;
29   String indexedListProperty;
30
31   public String getDepends() {
32       return this.depends;
33   }
34   public boolean isIndexed() {
35     return ((indexedListProperty != null
36                   && indexedListProperty.length() > 0));
37   }
38   public ValidatorResults validate(Map<String, Object> params, ... ) {
39     if (this.getDepends() == null) {
40        return new ValidatorResults();
41     }
42     ValidatorResults allResults = new ValidatorResults();
43     Object bean = params.get("java.lang.Object");
44
45     int numberOfFieldsToValidate = this.isIndexed() ?
46                    this.getIndexedPropertySize(bean) : 1;
47                              /* 2nd Invocation */
48
49     for (int i= 0; i< numberOfFieldsToValidate; fieldNumber++) {...}
50     return allResults;
51    }
52
53   String getIndexedListProperty() {
54       return this.indexedListProperty;
55   }
56
57   int getIndexedPropertySize(Object bean){ /* 3rd Invocation */
58       {...}
59       Object indexedProperty = PropertyUtils.getProperty(bean,
60            this.getIndexedListProperty()); /* vulnerable method */
61       {...}
62   }
63 }
```

Figure 2: A Library Affected by CVE-2019-10086

so it cannot resolve the case where the return value of other functions is used as the argument of the vulnerable function in its predefined migration rules, and thus fails to generate an exploit for F_1.

In the second function call (i.e., F_2 calls F_1 at Line 46), the argument `bean` passed to F_1 comes from `params` (Line 43). `params` is the first parameter of F_2, which is a `Map`; and the key for retrieving `bean` is `"java.lang.Object"`. Therefore, to exploit the vulnerability via F_2, `params` is required to set to {`"java.lang.Object"` : an instance of `Object`}, while preserving the program state when F_1 is called in the exploit generated in the previous step. Since the function argument is derived from a complex object (e.g., `Map`) with intricate data dependencies, and TRANSFER relies on similarity rules (e.g., string edit distance) to support simple data dependencies, it is unable to generate an exploit for F_2 to trigger the vulnerability in F_0.

In the first function call (i.e, F_3 calls F_2 at Line 12 and 19), it is required to invoke F_2 using the receiver object when F_2 is called in the exploit generated in the previous step. As this receiver object is acquired through the `lFields` and `hFields` fields in F_3, it must be put into these two fields. Furthermore, the data flow indicates that the parameter of F_3 is directly passed as the first argument to F_2. Thus, the first argument when F_2 is called in the exploit generated in the previous step can be used to pass to F_3. By this way, we can generate an exploit for F_3 to trigger the vulnerability in F_0.

The above procedure reveals the transitivity of exploitability along the call chain. Specifically, once we know the state of arguments and field values of the receiver object for F_1 to exploit a vulnerability, we only need to ensure that when F_2 executes and calls F_1, F_1 has the same state. In this way, we can exploit the vulnerability through F_2. Similarly, the same process applies to F_3. Given this property, we decompose a long call chain into an ordered sequence of call steps. For each call step, we use the exploit information from the previous call step to assist the current call step in exploiting the vulnerability, thereby significantly enhancing the success rate of generating an exploit in the complex scenarios.

Challenge 2: How to effectively satisfy the potentially complicated control flow and data flow conditions in a reachable call chain to exploit an vulnerability?

From the preceding analysis, we know that to generate an exploit for F_1, it is crucial to set the `indexedListProperty` field of the receiver object of F_1 to `"class"`. To generate an exploit for F_2, it is essential to ensure that the return values of `getDepends` (Line 39) and `isIndexed` (Line 45) meet the conditions for invoking F_1 (i.e., `getDepends` returns non-null and `isIndexed` returns true), while the argument to F_2 needs to meet specific conditions (e.g., `params` includes {`"java.lang.Object"` : an instance of `Object`}). When constructing an exploit for F_3, there are two branches that can lead to the invocation of F_2 (Line 8–13 and Line 14–22), but different branches need to satisfy different control flow and data flow conditions.

TRANSFER obtains the program state via a vulnerability-triggering test case. However, this program state only captures the condition at the vulnerable function call, and does not include the program state before calling the vulnerable function. Therefore, it heavily relies on EVOSUITE's search algorithm to satisfy the conditions of calling the vulnerable function. Similarly, VESTA defines search targets to guide EVOSUITE in generating test cases that call the vulnerable function. However, EVOSUITE can only handle simple path constraints. When dealing with complex scenarios such as the invocation of vulnerable function via functions like F_2 and F_3, EVOSUITE fails to generate test cases that reach the vulnerable function, resulting in false negatives.

Our initial attempt was to use symbolic execution for exploit generation. Symbolic execution achieves more accurate results but suffers from scalability issues with complex constraints (e.g., object combination constraints). Recent studies [12, 29, 40, 42, 55] have shown that large language models (LLMs) exhibit significant potential in understanding code semantics. LLMs trained on code can perform tasks such as program analysis, virus detection, and vulnerability identification, indicating that LLMs can not only capture the syntactic structure of code but also deeply understand its logic and intent (e.g., infer function states). Therefore, we attempt to leverage LLMs, combining static and dynamic analysis, to solve these complex constraints on the call chain.

3 APPROACH

Inspired by our motivating example in Sec. 2, we propose MAGNETO to automatically exploit vulnerabilities in dependent libraries.

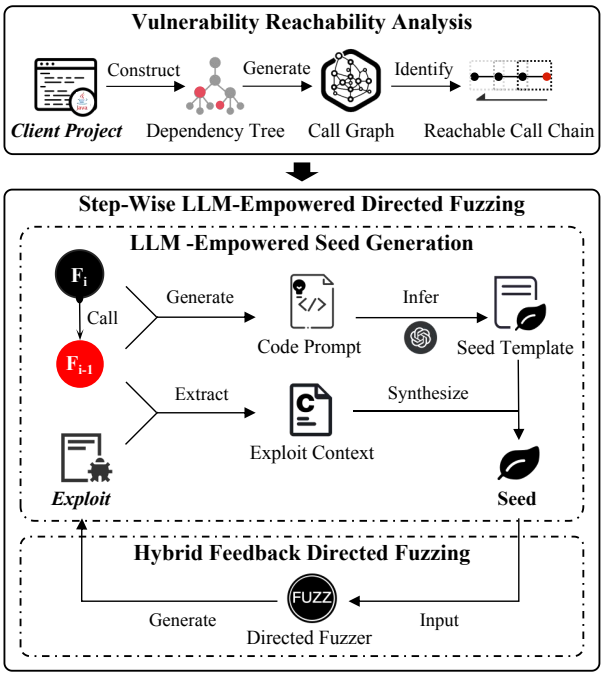

Figure 3: Approach Overview of Magneto

3.1 Approach Overview

Fig. 3 presents the approach overview of Magneto. It takes a client project as the input. Magneto works in five steps to exploit vulnerabilities in dependent libraries of the client project. First, it conducts a lightweight *vulnerability reachability analysis* to identify all call chains from the client project that can reach vulnerable functions (Sec. 3.2). Next, it conducts exploitability analysis on each reachable call chain $F_n \rightarrow F_{n-1} \rightarrow ... \rightarrow F_0$, where F_n is a function in the client project, and F_0 is the vulnerable function in the dependent library. It breaks down the call chain into a sequence of call steps in the reverse order, and conducts our *step-wise LLM-empowered directed fuzzing* on each call step $\langle F_i, F_{i-1} \rangle$ ($i = 1, ..., n$) individually. The final goal is to generate an exploit for F_n to trigger the vulnerability in F_0.

For each call step $\langle F_i, F_{i-1} \rangle$, the exploit for F_{i-1} is either generated in the previous call step analysis (when $i > 1$) or directly provided by a vulnerability database (when $i = 1$). Given F_{i-1} and the exploit for F_{i-1}, Magneto extracts the exploit context which records the required program state for F_{i-1} to exploit the vulnerability in F_0 (Sec. 3.3). This program state includes the receiver object on which F_{i-1} is called as well as the arguments passed to call F_{i-1}. Meanwhile, given F_i and F_{i-1}, Magneto uses prompt engineering to make LLM infer the seed template which records the required program state for F_i to call F_{i-1} (Sec. 3.4). This program state includes the receiver object on which F_i is called and the arguments passed to call F_i as well as their data flows to F_{i-1}. Subsequently, Magneto combines the exploit context and the seed template to synthesize a seed which is expected to execute F_i while preserving the required program state for F_{i-1} to exploit the vulnerability in F_0 (Sec. 3.5).

Due to the potentially complicated data flows to F_{i-1}, the synthesized seed may fail to preserve the required program state for F_{i-1}. Therefore, given the synthesized seed as the initial seed, Magneto finally leverages a directed fuzzer to mutate seeds such that the required program state for F_{i-1} becomes satisfied (Sec. 3.6). Our directed fuzzer utilizes hybrid feedback from the seed execution path to make better selections and mutations of seeds, ultimately exploiting the vulnerability in F_0. Then, the currently generated exploit assists in the exploitability analysis of the next call step.

Similar to existing approaches [10, 25, 28], Magneto is currently implemented for Java projects. However, our approach is generalizable, and we will extend it to support other programming languages.

3.2 Vulnerability Reachability Analysis

Given a client project, the goal of vulnerability reachability analysis is to identify reachable call chains to vulnerable functions through a lightweight dependency tree and call graph analysis.

Vulnerability Database. The vulnerability database provides the fundamental knowledge about vulnerabilities in dependent libraries [48, 49, 52]. It is required by all existing vulnerability existence, reachability and exploitation analysis methods. Many companies, including GitHub, GitLab, Veracode and Snyk, provide publicly available vulnerability databases. For each vulnerability, we collect the vulnerable dependencies (denoted as a 3-tuple of groupId, artifactId and version in Maven), the vulnerable function in each vulnerable dependency, the exploit for each vulnerable function, and the oracle for each exploit (which is used to validate whether the vulnerability is exploited). In summary, our database consists of a set of vulnerable dependencies V_{dep}, a set of vulnerable functions V_f, and a set of exploits V_{exp} with their corresponding oracles V_{oracle}. Here, we leverage V_{dep} and V_f in our vulnerability reachability analysis.

Vulnerability-Aware Dependency Tree Construction. Given the client project, Magneto uses the functionality provided by dependency managers to construct the dependency tree. Specifically, Magneto uses the Maven command `mvn dependency:tree` to obtain the dependency tree of the client project. To only consider dependencies that are used in the production environment [36], Magneto filters out dependencies with the scope `test`. Then, Magneto prunes the generated dependency tree by performing a depth-first search (DFS) starting from the root (i.e., the client project). When a vulnerable dependency in V_{dep} is encountered, the dependency chain from the root to this vulnerable dependency is recorded. After the DFS is complete, we only keep the dependencies in the recorded dependency chains and remove all other dependencies unrelated to the vulnerabilities. The resulting dependency tree is regarded as a vulnerability-aware dependency tree V_{tree}.

Call Graph Generation. Magneto downloads the corresponding JAR files for the dependencies in V_{tree} from the Maven repository, and generates the JAR file for the client project. Given these JAR files, Magneto leverages the class hierarchy analysis (CHA) [11] in Soot [19] to generate the call graph which contains call relations among the client project and the dependencies in V_{tree}.

Reachable Call Chain Identification. Given the generated call graph, Magneto identifies the call chains that reach vulnerable functions in V_f. Specifically, Magneto conducts a DFS on the generated call graph, starting from each public function in the client project. When a vulnerable function in V_f is encountered during the DFS, the call chain from the public function in the client project to this vulnerable function is recorded as a reachable call chain. After the

DFS is complete, MAGNETO identifies all the reachable call chains. We denote each reachable call chain as $F_n \rightarrow F_{n-1} \rightarrow ... \rightarrow F_0$, where F_n is the public function in the client project, and F_0 is the vulnerable function in V_f in the vulnerable dependency in V_{dep}.

3.3 Exploit Context Extraction

During our step-wise LLM-empowered directed fuzzing on each call step $\langle F_i, F_{i-1} \rangle$, the first step of MAGNETO is to extract the exploit context of F_{i-1} from the exploit for F_{i-1} (this exploit can be obtained from V_{exp} in vulnerability database). This exploit context records the required program state for F_{i-1} to exploit the vulnerability in F_0 through the call chain $F_{i-1} \rightarrow ... \rightarrow F_0$. Formally, we denote the exploit context of F_{i-1} as $C_{F_{i-1}}$, which is a 3-tuple $\langle F_{i-1}, A_{F_{i-1}}, R_{F_{i-1}} \rangle$. Here, $A_{F_{i-1}}$ denotes the arguments that are passed to call F_{i-1} in its exploit, and $R_{F_{i-1}}$ denotes the receiver object on which F_{i-1} is called in its exploit. Specifically, if F_{i-1} is a constructor or a static function, $R_{F_{i-1}}$ is $\emptyset$. Therefore, as long as we call F_{i-1} on $R_{F_{i-1}}$ and pass $A_{F_{i-1}}$ to its arguments, we can successfully exploit the vulnerability in F_0.

To extract the exploit context $C_{F_{i-1}}$, MAGNETO uses ASM [2] to automatically instrument code at the beginning of the function body of F_{i-1}. If F_{i-1} is a constructor or a static function, the instrumented code only records the passed arguments during the execution of F_{i-1}; otherwise, the instrumented code records both the passed arguments and the receiver object during the execution of F_{i-1}. Then, MAGNETO executes the exploit for F_{i-1}, and collects the arguments and receiver object when F_{i-1} is executed in the exploit.

Example. For the motivating example in Sec. 2, before performing fuzzing on the call step $\langle F_2, F_1 \rangle$, we first need to obtain the context C_{F_1} of F_1. By executing the exploit for F_1 which is generated in the analysis of the previous call step $\langle F_1, F_0 \rangle$, we can obtain $C_{F_1}.A_{F_1}$, which is an instance of `Object`, and $C_{F_1}.R_{F_1}$, which is a receiver object of F_1, with its `indexedListProperty` field set to `"class"`.

3.4 Seed Template Generation

During our step-wise LLM-empowered directed fuzzing on each call step $\langle F_i, F_{i-1} \rangle$, the second step of MAGNETO is to utilize LLM to infer how to set the the fields of the receiver object on which F_i is called and the arguments passed to call F_i so as to ensure that F_{i-1} is invoked in F_i. Theoretically, the receiver object and arguments to call F_i is are expressions over the receiver object and arguments to call F_{i-1} based on data flows from F_i to F_{i-1}, which is referred to as seed template, denoted as $T_{\langle F_i, F_{i-1} \rangle}$. It is challenging to scalably derive $T_{\langle F_i, F_{i-1} \rangle}$ by static analysis. Instead, we propose a prompt to instruct LLM to approximate this process and partially derive $T_{\langle F_i, F_{i-1} \rangle}$.

Figure 4 illustrates the prompt for generating the seed template $T_{\langle F_i, F_{i-1} \rangle}$. Specifically, [%CODE%] indicates the focal code that facilitates the understanding of how F_i calls F_{i-1}. [%CALLER%] denotes the name of the caller function (i.e., F_i), while [%CALLEE%] denotes the name of the callee function (i.e., F_{i-1}). [%ARGS%] and [%RO%] refers to the arguments and receiver object required to call F_{i-1}. In fact, [%ARGS%] and [%RO%] are extracted in Sec. 3.3, but to relieve the burden on LLM, we symbolize them, using [$\langle 0 \rangle$, $\langle 1 \rangle$, ..., $\langle n-1 \rangle$] to represent these arguments (where n is the number of arguments to call F_{i-1}) and $\langle RO \rangle$ to represent the receiver object. In addition, the JSON format of the returned $T_{\langle F_i, F_{i-1} \rangle}$ is defined in detail, which facilitates the subsequent processing of LLM's result.

Prompt Template

System: You are a professional Java code master. Just return the JSON body without additional explanation or comment.

Prompt Content:
[%CODE%]
In the provided code snippet, **[%CALLER%]** calls **[%CALLEE%]**. The objective is to set the fields of the receiver object and the arguments for **[%CALLER%]** to ensure that **[%CALLEE%]** is invoked with **[%ARGS%]** by **[%RO%]**. Note that **[%ARGS%]** and **[%RO%]** is just a list of symbol, representing the arguments and receiver object of **[%CALLEE%]**. You should use these symbols for the analysis.
The returned JSON must satisfy the following format:

```
{
    "FieldProperties" : {
       "${filed name}" : ${state},
       // Additional field properties as needed
    },
    "Arguments" : {
       "0" : ${state},
       // Additional argument as needed
    }
}
```

FieldProperties describes the conditions that must be satisfied by the fields of the receiver object of **[%CALLER%]**, and *Arguments* describes the conditions that must be satisfied by the arguments of **[%CALLER%]**.
FieldProperties and *Arguments* correspond to a JSON structure. ${field name} represents the name of the field of the receiver object. `"0"` represents the first argument. ${state} represents the value of the corresponding field or argument, which is in a JSON format of { type : "${type}", value : "${value}" }. ${type} would be a primitive type or the class name of an object.
If the type is a primitive type, such as int, char, ..., ${value} directly holds its value or symbol.
If the type is a reference type or Map type, ${value} is in a JSON format of a map.
If the type is an Array, Set or List type, ${value} corresponds to a JSON format of an array.

Figure 4: Prompt for Generating Seed Template

[%CODE%] plays a crucial role in the prompt, determining LLM's understanding of the data flow logic of calling F_{i-1} in F_i. Therefore, we propose an algorithm for generating focal code, aimed at maximally preserving the processing logic involved in the procedure of calling F_{i-1} in F_i, while trimming away unrelated code. It helps LLM avoid distractions from non-relevant code as well as token limit violations. We denote the focal code as the functions and fields of each class used during the procedure of calling F_{i-1} in F_i. Specifically, to avoid introducing classes not closely related to F_i, we only consider classes in the same dependency as the residing class of F_i.

First, MAGNETO uses JD-Core [18] to decompile the JAR file of the dependency which F_i belongs to, and then uses JavaParser [26] to trim F_i's code, only keeping the statements before the last call (as F_i

Response of LLM

```
"FieldProperties" : {
    "indexedListProperty" : {
        "type" : "java.lang.String",
        "value" : "Any non-empty string"
    }
},
"Arguments" : {
    "0" : {
        "type" : "java.util.Map",
        "value" : {
            "java.lang.Object" : {
                "type" : "java.lang.Object",
                "value " : "<0>"
            }
        ...
}
```

Figure 5: Response of the LLM in Generating Seed Template

may call F_{i-1} multiple times) to F_{i-1} and trimming the statements after the last call to F_{i-1}. We denote the resulting F_i as $\widetilde{F_i}$.

Next, MAGNETO leverages the functions and fields used in $\widetilde{F_i}$ for searching. We define two sets, *PF* and *PA*, denoting the functions and fields of the classes to be included in the focal code, respectively. MAGNETO first uses ASM [2] to extract the functions $UF_{\widetilde{F_i}}$ and fields $UP_{\widetilde{F_i}}$ used in $\widetilde{F_i}$, and adds them to *PF* and *PA*, respectively. At the same time, $UF_{\widetilde{F_i}}$ is used to initialize the search queue Q. Then, MAGNETO runs a breadth-first search (BFS) using Q. If Q is not empty, a function F_k is dequeued from Q, MAGNETO uses ASM to extract functions UF_{F_k} and fields UA_{F_k} used in F_k (only if F_k resides in the same dependency as F_i), adds them to *PF* and *PA*, and enqueues UF_{F_k} into Q. If Q is empty, our search process is complete. Notice that we set the maximum BFS search depth to 2. Specifically, each time a new UF_{F_k} is added to Q, the depth from F_i increases by one. As long as the depth does not exceed 2, our BFS continues; otherwise, our BFS stops. This prevents introducing massive classes that exceed the token limit of LLM, and keeps the relevance of identified code.

Finally, MAGNETO trims the code of the classes where the functions in *PF* and fields in *PA* reside, keeping only the functions in *PF* and the fields in *PA*. Then, MAGNETO concatenates them with $\widetilde{F_i}$ to form the focal code, and utilizes LLM with the complete prompt. LLM's response is the derived seed template $T_{\langle F_i, F_{i-1}\rangle}$.

Example. For the motivating example in Sec. 2, Fig. 5 shows the inferred seed template for the call step $\langle F_2, F_1\rangle$. In *FieldProperties*, LLM correctly infers that the `indexedListProperty` field of the receiver object of F_2 must be set to a non-empty string to ensure that `isIndexed` at Line 45 returns `true`, which is a prerequisite for invoking F_1 at Line 46. However, the `depends` field is not inferred by LLM, which must be set to non-null as required by the code at Line 39. This will be resolved by our directed fuzzer in Sec. 3.6. In *Arguments*, LLM correctly infers that the first argument of F_2 should be a map that contains a key-value instance. The key of this instance is `"java.lang.Object"`, and the value is an object that will be passed to F_1 as the first argument (represented by its symbol $\langle 0\rangle$).

3.5 Initial Seed Synthesis

After getting $C_{F_{i-1}}$ and $T_{\langle F_i, F_{i-1}\rangle}$, the third step of MAGNETO is to synthesize a seed that will be used as the inital seed for our directed fuzzing for the call step $\langle F_i, F_{i-1}\rangle$. We define the seed of F_i as S_{F_i}, which is a 4-tuple $\langle F_i, A_{F_i}, R_{F_i}, Score\rangle$, where F_i represents the target function that will be executed and fuzzed, A_{F_i} is the arguments passed to call F_i, R_{F_i} is the receiver object on which F_i is called, and $Score$ denotes the fitness score that will be introduced in Sec 3.6.

In fact, the seed template $T_{\langle F_i, F_{i-1}\rangle}$ which is inferred by LLM in a JSON format (e.g., Fig. 5) can be considered as a kind of serialization result of the expected $S_{F_i}.A_{F_i}$ and $S_{F_i}.R_{F_i}$. In that sense, we can use a kind of deserialization on $T_{\langle F_i, F_{i-1}\rangle}$ to instantiate $S_{F_i}.A_{F_i}$ and $S_{F_i}.R_{F_i}$. Specifically, MAGNETO uses *FieldProperties* in $T_{\langle F_i, F_{i-1}\rangle}$ to instantiate $S_{F_i}.R_{F_i}$. For fields listed in *FieldProperties*, it parses their inferred values and assigns them to the corresponding fields in $S_{F_i}.R_{F_i}$. Similarly, MAGNETO uses *Arguments* in $T_{\langle F_i, F_{i-1}\rangle}$ to instantiate $S_{F_i}.A_{F_i}$. For the i-th argument, it parses the inferred value and assigns it to the i-th argument in $S_{F_i}.A_{F_i}$. During the value parsing and assignment, the symbols $\langle 0\rangle$, $\langle 1\rangle$, ..., $\langle n-1\rangle$, $\langle RO\rangle$ in $T_{\langle F_i, F_{i-1}\rangle}$ is replaced by the corresponding objects in $C_{F_{i-1}}$; i.e., $\langle i\rangle$ is replaced by the i-th argument in $C_{F_{i-1}}.A_{F_{i-1}}$, and $\langle RO\rangle$ is replaced by $C_{F_{i-1}}.R_{F_{i-1}}$.

The currently synthesized $S_{F_i}.A_{F_i}$ and $S_{F_i}.R_{F_i}$ are inferred by LLM to expect that after seed execution, the receiver object and arguments passed to call F_{i-1} could satisfy the required program state (i.e., $C_{F_{i-1}}.A_{F_{i-1}}$) for exploiting the vulnerability. However, we observe that LLM is often not effective in handling [%RO%], and thus we use static analysis to complement $S_{F_i}.A_{F_i}$ and $S_{F_i}.R_{F_i}$.

Specifically, if F_i and F_{i-1} belong to the same class, MAGNETO traverses each field in $C_{F_{i-1}}.R_{F_{i-1}}$. If a field is not null, it assigns its value to the corresponding field in $S_{F_i}.R_{F_i}$. This is because such fields in the receiver object of F_{i-1} often participate in the vulnerability exploitation, and thus these fields need to be passed to the receiver object of F_i. However, when the exploitation requires the field to be null, this strategy might miss assigning the correct value to the field in $S_{F_i}.R_{F_i}$. To mitigate this issue, we rely on our directed fuzzing in Sec. 3.6 to correct such missed fields.

If F_i and F_{i-1} do not belong to the same class, in order to ensure that F_i can call F_{i-1} through $C_{F_{i-1}}.R_{F_{i-1}}$ during execution, we need to determine how $C_{F_{i-1}}.R_{F_{i-1}}$ is obtained in F_i. It might be obtained from a field of the receiver object of F_i or be one of F_i's arguments. Thus, MAGNETO marks the receiver object of F_{i-1} in F_i as tainted and uses inter-procedural field-sensitive backward taint analysis to identify the source of F_{i-1}'s receiver object in $S_{F_i}.A_{F_i}$ and $S_{F_i}.R_{F_i}$ (e.g., it may come from a field of $S_{F_i}.R_{F_i}$), and assigns $C_{F_{i-1}}.R_{F_{i-1}}$ to the corresponding source in $S_{F_i}.A_{F_i}$ and $S_{F_i}.R_{F_i}$.

Example. During the process of synthesizing the seed for the call step $\langle F_2, F_1\rangle$ in the motivating example in Sec. 2, MAGNETO synthesizes $S_{F_2}.R_{F_2}$ and $S_{F_2}.A_{F_2}$ by parsing the response from LLM (i.e., Fig. 5). When *FieldProperties* is parsed, the `indexedListProperty` field in $S_{F_2}.R_{F_2}$ is set to `"Any non-empty string"`. When *Arguments* is parsed, the first argument of $S_{F_2}.A_{F_2}$ is a `Map`, and one of its key-value instances is composed of `"java.lang.Object"` and $\langle 0\rangle$. Here, $\langle 0\rangle$ is the first argument of $C_{F_1}.A_{F_1}$, and is replaced by this argument. Moreover, as F_2 and F_1 belong to the same class (i.e., `Field`), the value of the non-null fields in $C_{F_1}.R_{F_1}$ is assigned to the corresponding field in $S_{F_2}.R_{F_2}$. As an another example, for the call step $\langle F_3, F_2\rangle$,

as F_3 and F_2 belong to different classes, MAGNETO identifies that F_2's receiver object comes from the `lFields` field. Thus, F_2's receiver object is assigned to the `lFields` field of F_3's receiver object.

3.6 Hybrid Feedback Directed Fuzzing

Given the synthesized seed S_{F_i} as the initial seed, MAGNETO performs directed fuzzing for the call step $\langle F_i, F_{i-1} \rangle$ to generate an exploit for F_i, i.e., to call F_{i-1} in F_i and exploit the vulnerability in F_0.

To improve the effectiveness and efficiency of directed fuzzing, we classify the variables (i.e., the receiver object A_{F_i} and the arguments R_{F_i}) in a seed into the following three categories.

- **Exploit-Related Variable.** These variables have a data flow to the receiver object and the arguments of the call to F_{i-1} during execution. We denote this set of variables as M_{exp}.
- **Control-Flow-Related Variable.** These variables control the program's execution path and branch decisions, thereby affecting the reachability of F_{i-1}. We denote this set of variables as M_{ctl}.
- **Other Variable.** These variables are irrelevant to branch conditions and exploitation. We denote this set of variables as M_{other}.

For exploit-related variables, we mark the receiver object and arguments of the call to F_{i-1} in F_i as tainted, and use inter-procedural field-insensitive backward taint analysis to identify the tainted variables in a seed and put them into M_{exp}. For control-flow-related variables, we mark all variables involved in control flows (e.g., variables used in `if` or `switch` statements) before calling F_{i-1} in F_i as tainted, and track the tainted variables in a seed and put them into M_{ctl}. Then, the remaining variables in a seed belong to M_{other}. Specifically, if a variable belongs to both M_{exp} and M_{ctl}, we consider this variable to belong only to M_{exp}. This information will be used as a kind of feedback for mutation to assist the directed fuzzing process.

Fitness Score of Seed. To effectively select and mutate seeds, we propose to evaluate and prioritize seeds based on feedback from the execution path. Specifically, MAGNETO first uses JaCoCo [17] to instrument F_i for collecting the line numbers covered by the seed execution, and uses this as feedback to guide the fuzzing process. If the line numbers covered by a seed execution are closer to the line number where F_{i-1} is called, the seed is considered more likely to reach F_{i-1}. We define this fitness score of a seed S by Eq. 1,

$$S.Score = \frac{\sum_{l \in \tau(S)} d\left(l, l_{F_{i-1}}\right)}{|\tau\left(S\right)|} \tag{1}$$

where $\tau(S)$ is the set of line numbers covered by the execution of the seed S, $l_{F_{i-1}}$ is the line number where F_{i-1} is called, and $d(l, l_{F_{i-1}})$ denotes the distance between a covered line number l in $\tau(S)$ and $l_{F_{i-1}}$.

A seed with a smaller fitness score will be prioritized for execution and mutation. Unlike traditional coverage-guided fuzzing, which aims to increase code coverage to trigger potential vulnerabilities [8], MAGNETO guides the fuzzer to focus more on how to execute the seed to reach F_{i-1} while exploiting the vulnerability in F_0, thereby generating an exploit for F_i more effectively.

Mutation Strategy. We employ different mutation strategies for different types of variables in a seed. In particular, if a variable belongs to M_{ctl}, we assign it a higher mutation probability to help the seed explore more branches, thereby increasing the likelihood of reaching the call to F_{i-1} in F_i. Here, the probability is set to 0.9. If a variable belongs to M_{exp}, which typically contains specific knowledge for exploiting the vulnerability and is often well constructed in our seed synthesis based on the exploit context, we assign it a lower mutation probability of 0.1. If a variable belongs to M_{other}, we do not mutate it because it is not related to vulnerability exploitation. In this way, our fuzzer can understand the contribution of different variables to the exploit and perform mutations accordingly.

MAGNETO sequentially mutates each variable in M_{ctl} and M_{exp} according to their mutation probability. Once a variable is to be mutated, we mutate it based on its type. Specifically, for a primitive type (e.g., `int`), our fuzzer generates a new value via a random method (e.g., `random.nextInt()`). For a reference type, our fuzzer has a 0.5 chance of selecting an subclass of the given class for instantiation and initializing the fields with random values; and there is also a 0.5 chance that our fuzzer assigns a random value to one field of the variable. For an array type, our fuzzer has a 0.5 chance of creating a new array object. It first employs `random.nextInt()` to determine the array size and then assigns random values to the array elements based on their types. There is also a 0.5 chance that our fuzzer mutates one element in the existing array according to its type.

Oracle Analysis. We leverage ASM [2] to instrument the function body of the vulnerable function F_0 in order to record its execution status, i.e., its return value and thrown exception, during the execution of a seed. After each seed execution, MAGNETO checks if F_0 is executed. If F_0 is not executed, the seed has failed to exploit the vulnerability, and MAGNETO proceeds to the next round of mutation and evaluation. If F_0 is executed, MAGNETO feeds its recorded return value and thrown exception into the vulnerability oracle (which can be obtained from V_{oracle} in vulnerability database) for assessment. If the oracle passes, the vulnerability has not been exploited, and MAGNETO continues to the next round. If the oracle fails, it indicates that the vulnerability has been exploited. Therefore, MAGNETO has successfully generates the exploit for F_i.

Example. Recall that in the generated seed template for the call step $\langle F_2, F_1 \rangle$, invoking F_1 in F_2 also requires `getDepends` to return a non-null string, but LLM fails to infer this, preventing our synthesized seed from directly exploiting the vulnerability. This is compensated by our directed fuzzer through mutating the `depends` field of the receiver object in the seed, successfully correcting this issue and exploiting the vulnerability in F_0 through F_2.

4 EVALUATION

4.1 Evaluation Setup

To evaluate the effectiveness and efficiency of MAGNETO in generating vulnerability exploits for real-world projects, we design three research questions. We run our experiments on a Linux workstation with an Intel(R) Xeon(R) Silver 4316@2.30GHz and 256 GB of RAM, running Ubuntu 22.04.4 LTS with JDK 1.8.0 412.

- **RQ1 Effectiveness Evaluation:** How is the effectiveness of MAGNETO in generating exploits for vulnerabilities in libraries?
- **RQ2 Efficiency Evaluation:** What is the time overhead of MAGNETO in generating exploits for vulnerabilities in libraries?
- **RQ3 Ablation Study:** What is the contribution of the design decisions in MAGNETO to the effectiveness of MAGNETO?

Vulnerability Database Collection. To ensure our vulnerability database covers a wide range of vulnerability types, we gather

vulnerabilities in NVD from 2015 to 2023 which have patch-like links (e.g., commits in GitHub repositories) in their references. We collect a total of 942 vulnerabilities. Then, we focus on Java vulnerabilities, and locate their library repositories and verify the correctness of their patches based on the patch-like links. Next, we only keep the vulnerabilities whose patches include test cases, as these test cases often contain information related to vulnerability exploitation. Then, based on the test cases and modification information in the patches, we identify the corresponding vulnerable functions, and write the corresponding exploits in the form of JUnit test cases. In total, we collect 28 Java vulnerabilities with their exploits. In addition, we supplement our database with the vulnerabilities used in TRANSFER [28] and VESTA [10]. 6 of the 22 vulnerabilities in TRANSFER and 14 of the 30 vulnerabilities in VESTA are included, while others are not included because we fail to manually exploit them. Ultimately, we collect 32 Java vulnerabilities (and 45 vulnerable functions) with their exploits from 21 libraries, covering 17 types of vulnerabilities (i.e., CWE). They affect a wide range of libraries, e.g., JSON, PDF, archive processing, HTTP framework, and data manipulation tools.

We also need to get the version ranges of the libraries affected by these vulnerabilities. This information is often disclosed in NVD, but it may contain significant false positives or false negatives. Thus, we decide to run the exploit on all versions of the library, if the vulnerability is successfully exploited, we consider that version to be affected. To verify whether a vulnerability is exploited by an exploit, we create its oracle based on the exploit for the vulnerability. When the oracle is violated, it indicates that the vulnerability has been exploited [51]. This allows us to get a more precise list of affected library versions. Finally, we collect 970 affected library versions.

State-of-the-Art. To the best of our knowledge, we select all the state-of-the-art tools in this direction, i.e., SIEGE [25], TRANSFER [28] and VESTA [10]. Except for SIEGE, all tools, including ours, require the exploit at the library side as the tool input.

RQ1 Setup. We compare MAGNETO with the state-of-the-art tools in evaluating the effectiveness in generating vulnerability exploits. To comprehensively assess the effectiveness of each tool in complex scenarios involving reachable call chains, we evaluate not only the number of vulnerabilities each tool can exploit for client projects but also the capability of each tool in generating exploits through reachable call chains. To this end, we manually construct a ground truth for each pair of a vulnerability and a client project, and measure each tool's effectiveness based on multiple metrics. To evaluate each tool on a same baseline, our metrics describe the effectiveness in exploiting one vulnerability for one client project (i.e., the vulnerability database only contains the vulnerability under exploiting).

RQ2 Setup. We measure the average time taken by MAGNETO and the state-of-the-art tools to generate the exploit for each pair of a vulnerability and a client project. Notice that, the state-of-the-art tools do not have the vulnerability reachability analysis component, but assume the public function F_n in the client project is given as an input. For a fair comparison, we use our vulnerability reachability analysis component as the first step of the state-of-the-art tools.

RQ3 Setup. We create several ablated versions of MAGNETO to evaluate the design decisions made in MAGNETO. First, as the original version of MAGNETO uses GPT-4, we create a version that uses GPT-3 to evaluate the impact of different LLMs. Second, we create two versions by removing seed template generation and removing directed fuzzing to investigate how they contribute to MAGNETO. Third, we also create a version by removing the static analysis in seed synthesis, to investigate how it contributes to MAGNETO. Finally, we evaluate the sensitivity of MAGNETO to the BFS search depth in seed template generation by setting it to 1, 2, 3 and 4 respectively.

Ground Truth Construction. We search from GitHub projects that call vulnerable functions of our collected vulnerabilities and whose dependent library versions fall into the affected version range. Then, for each client project, we first use static analysis tools to construct its call graph, and identify the reachable call chains to the vulnerable functions. Next, we manually construct the exploit for each reachable call chain to verify that the vulnerability can be exploited through the call chain. To address potential biases, three experts with four years of experience in Java security were involved. Two experts were asked to try their best to independently identify exploitable call chains for each project. Then, the third expert conducted a final review of their findings, with the Cohen's Kappa coefficient [47] being 0.962. Finally, we collect 84 pairs of vulnerability and client project, and 182 exploitable call chains in 49 projects with a total length of 429. 78 exploitable call chains have a length of one. The average length of all exploitable call chains is 2.36, with the longest exploitable call chain consisting of 8 calls. The entire construction takes around one person-month.

Effectiveness Metrics. We measure the effectiveness using four metrics. Specifically, the *EPN* metric indicates the number of pairs of vulnerability and client project where a tool successfully generates an exploit for at least one exploitable call chain. The *ECN* metric indicates the number of exploitable call chains for which a tool successfully generates an exploit. The *MEL* and *AEL* metrics respectively measure the maximum and average length of the exploited call chains across all the exploitable call chains (if an exploitable call chain is not exploited, its length is counted as zero).

4.2 Effectiveness Evaluation (RQ1)

Overall Results. Table 1 reports the results of our effectiveness evaluation by comparing MAGNETO with all the state-of-the-art tools. SIEGE is not reported in Table 1 because it fails to exploit any vulnerability. In terms of *EPN*, MAGNETO successfully generates an exploit for 79 (94.0%) pairs of vulnerability and project, which outperforms TRANSFER and VESTA by 75.6% and 97.5%, respectively. In terms of *ECN*, MAGNETO successfully generates an exploit for 135 (74.2%) exploitable call chains, which improves TRANSFER and VESTA by 154.7% and 206.8%. In terms of *MEL* and *AEL*, MAGNETO achieves a maximum length of 6 and an average length of 1.47, significantly outperforming the best of the state-of-the-art by 100.0% and 308.3%.

Moreover, as MAGNETO is step-wise, we also investigate those partially exploited call chains (i.e., MAGNETO can generate an exploit for F_i $(0 < i < n)$ but fails for F_{i+1}). Specifically, MAGNETO partially generates an exploit for 42 exploitable call chains, meaning that it fails for the first call step $\langle F_1, F_0 \rangle$ for only 5 exploitable call chains.

Reasons for Failures. We summarize two main reasons for MAGNETO's failure in generating exploits. First, if the extracted focal code contains too many functions due to complex functionalities, LLM becomes overwhelmed. This makes it difficult for LLM to understand the semantics and hence unable to make correct inferences. On the contrary, if many of the used functions belong to third-party

Table 1: Results of Our Effectiveness Evaluation Compared to the State-of-the-Art

Affected Library	Vulnerability	Ground Truth				Transfer				Vesta				Magneto			
		EPN	ECN	MEL	AEL	EPN	ECN	MEL	AEL	EPN	ECN	MEL	AEL	EPN	ECN	MEL	AEL
Dom4j	CVE-2018-1000632	1	7	1	1.00	1	7	1	1.00	1	7	1	1.00	1	7	1	1.00
Xstream	CVE-2017-7957	2	4	1	1.00	1	1	1	0.25	2	3	1	0.75	2	4	1	1.00
	CVE-2020-26217	5	7	2	1.14	5	6	1	0.86	4	6	1	0.86	4	6	1	0.86
	CVE-2021-43859	6	8	2	1.12	1	1	1	0.12	6	7	2	1.00	5	7	1	0.88
	CVE-2022-41966	6	8	2	1.12	5	5	1	0.62	5	7	1	0.88	5	7	1	0.88
Spring Security	CVE-2020-5408	3	4	1	1.00	2	2	1	0.50	0	0	0	0.00	3	4	1	1.00
	CVE-2022-22976	2	2	1	1.00	2	2	1	1.00	0	0	0	0.00	2	2	1	1.00
JSON Sanitizer	CVE-2020-13973	2	4	4	2.75	2	2	3	1.00	1	0	0	0.00	2	2	3	1.00
	CVE-2021-23899	2	4	4	2.75	2	1	1	0.25	1	0	0	0.00	2	2	3	1.00
	CVE-2021-23900	2	4	4	2.75	2	1	1	0.25	1	0	0	0.00	2	2	3	1.00
Junrar	CVE-2022-23596	1	1	2	2.00	0	0	0	0.00	1	0	0	0.00	1	1	2	2.00
Jodd HTTP	CVE-2022-29631	3	5	3	2.20	2	1	1	0.20	2	2	1	0.40	3	5	3	2.20
HTTP Components	CVE-2020-13956	4	11	5	4.27	2	2	3	0.55	2	0	0	0.00	4	11	5	4.27
Fastjson	CVE-2022-25845	3	3	2	1.33	0	0	0	0.00	0	0	0	0.00	2	1	1	0.33
Snappy Java	CVE-2023-34453	1	1	1	1.00	0	0	0	0.00	0	0	0	0.00	1	1	1	1.00
	CVE-2023-43642	1	1	1	1.00	0	0	0	0.00	0	0	0	0.00	1	1	1	1.00
Commons Compress	CVE-2018-1324	2	2	3	2.00	0	0	0	0.00	1	0	0	0.00	2	1	1	0.50
	CVE-2021-35516	3	5	6	2.40	0	0	0	0.00	1	1	1	0.20	3	4	2	1.20
Zt Zip	CVE-2018-1002201	2	2	3	2.00	0	0	0	0.00	0	0	0	0.00	2	2	3	2.00
Netty	CVE-2015-2156	1	6	5	4.00	0	0	0	0.00	0	0	0	0.00	1	0	0	0.00
JSON	CVE-2022-45688	2	5	2	1.60	0	0	0	0.00	0	0	0	0.00	2	3	2	0.80
Jackson Databind	CVE-2022-42004	4	4	1	1.00	0	0	0	0.00	0	0	0	0.00	4	4	1	1.00
Commons IO	CVE-2021-29425	3	9	3	2.22	2	3	1	0.33	2	1	1	0.11	3	7	3	1.56
	IO-611	3	9	3	2.22	3	5	3	1.00	2	1	1	0.11	3	9	3	2.22
Spring Web	CVE-2018-15756	1	1	2	2.00	0	0	0	0.00	0	0	0	0.00	0	0	0	0.00
Commons Lang3	LANG-1385	2	3	2	1.33	2	3	2	1.33	1	2	1	0.67	2	3	2	1.33
	LANG-1645	2	4	2	1.50	2	3	2	1.00	1	2	1	0.50	2	4	2	1.50
PDF Box	CVE-2021-31812	2	37	6	3.32	0	0	0	0.00	1	1	1	0.03	2	21	6	2.03
Zip4j	Zip4j-263	2	2	1	1.00	2	2	1	1.00	2	2	1	1.00	2	2	1	1.00
Commons Beanutils	CVE-2019-10086	3	9	8	4.67	3	1	1	0.11	0	0	0	0.00	3	2	2	0.33
JSON Smart	CVE-2021-27568	3	4	2	1.25	3	4	2	1.25	1	1	1	0.25	3	4	2	1.25
	CVE-2023-1370	5	6	2	1.17	1	1	1	0.17	2	1	1	0.17	5	6	2	1.17
Total 21	**32**	**84**	**182**	**8**	**2.36**	**45**	**53**	**3**	**0.36**	**40**	**44**	**2**	**0.25**	**79**	**135**	**6**	**1.47**

Table 2: Effectiveness Results w.r.t. the Length of Call Chains

Call Chain Length	Ground Truth	Transfer	Vesta	Magneto
1	78	45	43	77
2	29	3	1	15
3	37	5	0	22
4	18	0	0	12
5	13	0	0	8
6	5	0	0	1
7	1	0	0	0
8	1	0	0	0

dependencies and thus are not extracted for the focal code, LLM has too little code context to make correct inferences. Meanwhile, our static analysis is also ineffective in handling such complex scenarios. As a result, Magneto fails to generate exploits. Second, some arguments must originate from external environments, e.g., HTTP requests. We cannot instantiate these objects using new or reflection, making it impossible to exploit such vulnerabilities.

Length Distribution of Exploited Call Chains. The first two columns of Table 2 list the length distribution of the 182 exploitable call chains in our ground truth, and the last three columns report the length distribution of the successfully exploited call chains for each tool. For these 78 exploitable call chains with a length of one, Transfer and Vesta demonstrate relatively good effectiveness. Magneto exhibits a strong performance in these scenarios, which only fails in one case. For the exploitable call chains with a length lager than one, Transfer and Vesta experience a sharp decline in performance, and only succeed in a few cases. Magneto maintains a relatively good performance in these scenarios thanks to our step-wise design.

Table 3: Results of Our Efficiency Evaluation

Magneto			Siege	Transfer	Vesta
Vul. Reach. Analysis	Directed Fuzzing	Total			
1.6 m	9.3 m	10.9 m	96.4 m	10.4 m	6.3 m

Summary: Magneto significantly outperforms all the state-of-the-art tools, achieving an improvement of at least 75.6% and 154.7% in terms of *EPN* and *ECN*. In addition, Magneto demonstrates a relatively good performance for exploitable call chains with a length larger than one, where all the state-of-the-art tools work poorly. These results demonstrate the effectiveness of Magneto in generating exploits.

4.3 Efficiency Evaluation (RQ2)

Table 3 lists the results of our efficiency evaluation. Magneto takes an average of 10.9 minutes to analyze each pair of vulnerability and project. Specifically, vulnerability reachability analysis costs 1.6 minutes (which is the same across all tools), while the whole directed fuzzing costs 9.3 minutes. Siege has the highest time overhead of 96.4 minutes, and Vesta has the lowest (i.e., 6.3 minutes).

Summary: It takes 10.9 minutes for Magneto to generate exploits for each pair of vulnerability and project, which is 4.6 minutes slower than the fastest. This is acceptable due to our significant improvement in effectiveness.

Table 4: Results of Our Ablation Study

Ablated Version	Single-Step Scenario				Multi-Step Scenario			
	EPN	ECN	AEL	MEL	EPN	ECN	AEL	MEL
GPT-4-turbo (BFS Search Depth = 2)	56	77	0.99	1	31	58	1.83	6
GPT-3-turbo	52	73	0.94	1	28	39	1.25	5
w/o Seed Template Generation	54	75	0.96	1	24	13	0.33	3
w/o Directed Fuzzing	52	72	0.92	1	26	33	1.04	5
w/o Static Analysis in Seed Synthesis	49	63	0.81	1	22	34	1.00	5
BFS Search Depth = 1	55	76	0.97	1	28	32	0.79	3
BFS Search Depth = 3	55	75	0.96	1	30	34	0.86	4
BFS Search Depth = 4	55	75	0.96	1	28	31	0.89	4

4.4 Ablation Study (RQ3)

Table 4 shows the ablation results. The second row reports the original version of Magneto that uses GPT-4 and set BFS depth to 2.

Impact of LLMs. After changing GPT-4 to GPT-3, Magneto experienced a drop of 7.1% in *EPN*, 5.2% in *ECN*, and 5.1% in *AEL* in the Single-Step scenario, and a drop of 9.7% in *EPN*, 32.8% in *ECN*, 31.7% in *AEL*, and 16.7% in *MEL* in the Multi-Step scenario. These results indicate that the capability of LLMs affects the quality of inferred seed templates, especially in the Multi-Step scenario.

Impact of Employed Strategies. After removing seed template generation (i.e., relying on the static analysis in seed synthesis to derive the seed), Magneto suffers a significant decrease in all the metrics, especially in *ECN* (by 77.6%), *AEL* (by 82.0%) and *MEL* (by 50.0%) in the Multi-Step scenario. After removing directed fuzzing (i.e., directly using the synthesized seeds to exploit the vulnerability), Magneto has a large decline in all the metrics, especially in *ECN* (by 43.1%) in the Multi-Step scenario. After removing the static analysis in seed synthesis, Magneto suffers a great drop in all the metrics, especially in *EPN* (by 29.0%), *ECN* (by 41.4%) and *AEL* (by 45.4%) in the Multi-Step scenario. These results highlight the value of each strategy and the rationality of their combination.

Impact of Focal Code Scope. We configure the BFS search depth that is used to determine the scope of extracted focal code. A threshold of 2 yields the best results across the four metrics in both Single-Step and Multi-Step scenarios. Both a smaller and larger scope hinder LLM's inference capability.

Summary: Magneto relies on the capability of LLM as well as the scope of focal code to better derive seed templates. Besides, all the strategies have an important contribution to the Magneto. The results also indicate that static and dynamic analyses have a good performance on Single-Step scenarios, while using LLMs is more effective in Multi-Step scenarios.

4.5 Discussion

Threats. A primary threat to our evaluation is the construction of the ground truth. We search for affected client projects based on the vulnerabilities in our vulnerability database, which might limit the range of client projects we find. Consequently, there is a possibility that we may miss client projects with different or newer types of vulnerabilities not present in our vulnerability database. However, we endeavor to collect as many vulnerabilities as possible and seek as many projects as possible where vulnerabilities can be exploited via reachable call chains, hence enhancing the diversity and complexity of our dataset. Compared to the ground truth benchmarks of Transfer and Vesta, our ground truth is the largest, and includes more complex vulnerability exploitation scenarios, thereby providing a more comprehensive evaluation of Magneto's effectiveness. In addition, Magneto may be affected by LLM data leakage. To mitigate this issue, our ground truth collection process focuses on selecting recently updated projects. Moreover, while LLMs may be trained on related code, they were not trained for this specific task.

Limitations. First, Magneto relies on a vulnerability database, and the scope of it may impact Magneto's effectiveness. To address this issue, we have implemented an semi-automated pipeline that crawls vulnerability patches and extracts the affected libraries and related test cases. Using these test cases and patches, we can gather exploits and continuously update the database. Second, Magneto uses GPT as the LLM to infer seed templates. However, GPT is closed-source. We plan to evaluate Magneto with different open-source LLMs. Third, Magneto faces challenges in handling particularly complex object combinations when using LLM to infer seed template. To partially mitigate this, we leverage static analysis techniques to complement seed template, and achieving the best results in our experiments compared to state-of-the-art tools.

5 RELATED WORK

We review the most closely related work in three areas, i.e., exploit generation, third-party library security, and fuzz testing.

5.1 Exploit Generation

An exploit refers to a piece of code that takes advantage of a vulnerability in an application to cause unintended behavior. It plays a crucial role in assessing the severity and security risks of vulnerabilities. There are two lines of work that aim to automatically generate exploits. On the one hand, some researchers (e.g., [3, 6, 46]) focus on generating an exploit for a vulnerability in a program. For example, Brumley et al. [6] proposed to use patches to generate exploits for input-validation vulnerabilities. Avgerinos et al. [3] leveraged symbolic execution to generate control-flow hijack exploits. Wang et al. [46] introduced a layout-oriented fuzzing and control-flow stitching approach to generate exploits for heap-based vulnerabilities.

On the other hand, some researchers [10, 25, 28] concentrate on generating an exploit for a vulnerability in a dependent library of a project. Their purpose is to assess whether the vulnerability in the dependent library can be triggered by the project. Iannone et al. [25] proposed the first approach Siege in this direction. Siege uses the vulnerable code in the dependent library as the search goal, and employs search-based testing approach EvoSuite [14] to generate test cases for the project to execute the vulnerable code, thereby evaluating the exploitability of the vulnerability. To improve the likelihood of generating such test cases, Kang et al. [28] introduced Transfer and Chen et al. [10] developed Vesta to utilize the knowledge of the vulnerability-witnessing test case in the dependent library. Specifically, Transfer executes the vulnerability-witnessing test case to capture the program state of the vulnerable function at the library side. Then, it uses the program state to guide EvoSuite to generate test cases of the project that invoke the vulnerable function with the same program state, thereby creating an exploit that triggers the vulnerability from the project. Differently, Vesta first utilizes EvoSuite to generate test cases of the project that reach the vulnerable

function. Then, according to predefined rules, it migrates parameters from the vulnerability-witnessing test case into these generated test cases in order to trigger the vulnerability.

The second line of work [10, 25, 28] is the closest to ours. However, these tools only handle simple scenarios where the vulnerable function is directly invoked by the project, but often fail to deal with complex scenarios where the vulnerable function is indirectly invoked along a long call chain, leading to false negatives. Instead, MAGNETO performs directed fuzzing on each function in the call chain, and uses the information from the previous function's vulnerability exploit to assist in the next function.

5.2 Third-Party Library Security

The reuse of third-party libraries significantly speeds up the development process of software projects. However, this convenience also introduces substantial security risks. To gain a more comprehensive understanding of the security risks posed by vulnerabilities in third-party libraries, several empirical studies have been conducted in various ecosystems. For example, Zhan et al. [56] reported that 74.95% of the vulnerable third-party libraries were widely used by other libraries in the Android ecosystem. Hu et al. [22] found that 66.10% of the downstream projects in the Go ecosystem were affected by vulnerabilities in their dependencies, with 62.85% of these projects yet to address these issues. Liu et al. [31] discovered that 19.96% of the libraries were transitively affected by 416 vulnerabilities in the NPM ecosystem. These analyses determine whether a project is affected by a vulnerability based on whether the project depends on the vulnerable library, without considering whether the project actually invokes the vulnerable function.

To reduce false positives of previous analyses, several approaches have been proposed to conduct reachability analysis of vulnerabilities. For example, Wang et al. [24, 45] leveraged call graph analysis to determine whether a vulnerability could be reached by a project. Wu et al. [50] found that 86.1% of the vulnerable functions were not actually reachable by their downstream projects. Xu et al. [53] proposed to support vulnerability reachability analysis across different ecosystems. However, as these analyses do not consider whether the vulnerability can be actually triggered, many false positives still remain. MAGNETO is designed to further reduce false positives by assessing the exploitability of the vulnerability.

5.3 Fuzz Testing

A lot of fuzz testing approaches (e.g., [4, 5, 7, 9, 13, 21, 35, 41, 43, 44, 54]) have been proposed to discover bugs or vulnerabilities in programs. Some approaches (e.g., [5, 13, 35, 41, 54]) utilized instrumentation and static analysis to maximize branch coverage, thereby increasing the probability of finding vulnerabilities. However, they are limited when the programs under test deal with structured inputs (e.g., XML or code). Therefore, several approaches (e.g., [21, 43, 44]) proposed grammar-aware fuzzers that generate structured inputs based on their grammars. Unfortunately, they may not be effective in scenarios such as patch presence testing or vulnerability reproduction testing, which are required to target specific code snippets. Hence, directed fuzzers (e.g., [4, 7, 9]) have been proposed to guide the generated inputs to reach specific code snippets. MAGNETO also utilizes directed fuzzing but for a different task of automated exploit generation for library vulnerabilities.

6 CONCLUSIONS

In this paper, we have proposed a novel step-wise approach, named MAGNETO, to exploit vulnerabilities in dependent libraries for client projects through LLM-empowered directed fuzzing. Our evaluation has demonstrated the effectiveness and efficiency of MAGNETO. In the future, we plan to apply MAGNETO to analyze more downstream projects to reduce their security risks, and extend MAGNETO to support other programming languages.

ACKNOWLEDGMENT

This work was supported by the National Natural Science Foundation of China (Grant No. 62332005 and 62372114).

REFERENCES

[1] Apache. 2024. *Update Apache Commons BeanUtils dependency from 1.9.3 to 1.9.4.* Retrieved May 28, 2024 from https://issues.apache.org/jira/browse/VALIDATOR-460
[2] ASM. 2024. *ASM.* Retrieved May 30, 2024 from https://asm.ow2.io/
[3] 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.
[4] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*. 2329–2344.
[5] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 1032–1043.
[6] David Brumley, Pongsin Poosankam, Dawn Song, and Jiang Zheng. 2008. Automatic patch-based exploit generation is possible: Techniques and implications. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*. 143–157.
[7] Sicong Cao, Biao He, Xiaobing Sun, Yu Ouyang, Chao Zhang, Xiaoxue Wu, Ting Su, Lili Bo, Bin Li, Chuanlei Ma, et al. 2023. Oddfuzz: Discovering java deserialization vulnerabilities via structure-aware directed greybox fuzzing. In *Proceedings of the 2023 IEEE Symposium on Security and Privacy*. 2726–2743.
[8] Sicong Cao, Xiaobing Sun, Xiaoxue Wu, Lili Bo, Bin Li, Rongxin Wu, Wei Liu, Biao He, Yu Ouyang, and Jiajia Li. 2023. Improving java deserialization gadget chain mining via overriding-guided object generation. In *Proceedings of the 45th International Conference on Software Engineering*. 397–409.
[9] 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 *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*. 2095–2108.
[10] Zirui Chen, Xing Hu, Xin Xia, Yi Gao, Tongtong Xu, David Lo, and Xiaohu Yang. 2024. Exploiting Library Vulnerability via Migration Based Automating Test Generation. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–12.
[11] Jeffrey Dean, David Grove, and Craig Chambers. 1995. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the 9th European Conference on Object-Oriented Programming*. 77–101.
[12] Chongzhou Fang, Ning Miao, Shaurya Srivastav, Jialin Liu, Ruoyu Zhang, Ruijie Fang, Asmita Asmita, Ryan Tsang, Najmeh Nazari, Han Wang, et al. 2023. Large language models for code analysis: Do llms really do their job? (2023).
[13] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. {AFL++}: Combining incremental steps of fuzzing research. In *Proceedings of the 14th USENIX Workshop on Offensive Technologies*.
[14] 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*. 416–419.
[15] Github. 2024. *Apache Commons Beanutils.* Retrieved May 28, 2024 from https://github.com/apache/commons-beanutils
[16] Github. 2024. *Apache Commons Validator.* Retrieved May 28, 2024 from https://github.com/apache/commons-validator
[17] Github. 2024. *Java Code Coverage Library.* Retrieved May 28, 2024 from https://github.com/jacoco/jacoco
[18] Github. 2024. *jd-core.* Retrieved May 30, 2024 from https://github.com/java-decompiler/jd-core
[19] Github. 2024. *Soot - A Java optimization framework.* Retrieved May 29, 2024 from https://github.com/soot-oss/soot

[20] Google. 2024. *Understanding the Impact of Apache Log4j Vulnerability.* Retrieved May 26, 2024 from https://security.googleblog.com/2021/12/understanding-impact-of-apache-log4j.html

[21] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with code fragments. In *Proceedings of the 21st USENIX Security Symposium*. 445–458.

[22] Jinchang Hu, Lyuye Zhang, Chengwei Liu, Sen Yang, Song Huang, and Yang Liu. 2024. Empirical Analysis of Vulnerabilities Life Cycle in Golang Ecosystem. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.

[23] Kaifeng Huang, Bihuan Chen, Linghao Pan, Shuai Wu, and Xin Peng. 2021. REPFINDER: Finding replacements for missing APIs in library update. In *Proceedings of the 2021 IEEE/ACM 36th International Conference on Automated Software Engineering*. 266–278.

[24] Kaifeng Huang, Bihuan Chen, Congying Xu, Ying Wang, Bowen Shi, Xin Peng, Yijian Wu, and Yang Liu. 2022. Characterizing usages, updates and risks of third-party libraries in Java projects. *Empirical Software Engineering* 27, 4 (2022), 90.

[25] Emanuele Iannone, Dario Di Nucci, Antonino Sabetta, and Andrea De Lucia. 2021. Toward automated exploit generation for known vulnerabilities in open-source libraries. In *Proceedings of 2021 IEEE/ACM 29th International Conference on Program Comprehension*. 396–400.

[26] JavaParser. 2024. *JavaParser Home.* Retrieved May 30, 2024 from https://javaparser.org/

[27] Dhanushka Jayasuriya, Valerio Terragni, Jens Dietrich, Samuel Ou, and Kelly Blincoe. 2023. Understanding Breaking Changes in the Wild. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1433–1444.

[28] 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*. 276–288.

[29] Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. 2024. Enhancing Static Analysis for Practical Bug Detection: An LLM-Integrated Approach. *Proceedings of the ACM on Programming Languages* 8, OOPSLA1 (2024), 474–499.

[30] Wenke Li, Feng Wu, Cai Fu, and Fan Zhou. 2023. A Large-Scale Empirical Study on Semantic Versioning in Golang Ecosystem. In *Proceedings of the 2023 IEEE/ACM 38th International Conference on Automated Software Engineering*. 1604–1614.

[31] Chengwei Liu, Sen Chen, Lingling Fan, Bihuan Chen, Yang Liu, and Xin Peng. 2022. Demystifying the vulnerability propagation and its evolution via dependency trees in the npm ecosystem. In *Proceedings of the 44th International Conference on Software Engineering*. 672–684.

[32] Samim Mirhosseini and Chris Parnin. 2017. Can automated pull requests encourage software developers to upgrade out-of-date dependencies?. In *Proceedings of the 32nd IEEE/ACM international conference on Automated Software Engineering*. 84–94.

[33] 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*. 29–41.

[34] NVD. 2024. *CVE-2019-10086 Detail.* Retrieved May 28, 2024 from https://nvd.nist.gov/vuln/detail/CVE-2019-10086

[35] Rohan Padhye, Caroline Lemieux, and Koushik Sen. 2019. Jqf: Coverage-guided property-based testing in java. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 398–401.

[36] 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* 48, 5 (2020), 1592–1609.

[37] 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 *Proceedings of the IEEE International Conference on Software Maintenance and Evolution*. 449–460.

[38] sonatype. 2023. *9th Annual State of the Software Supply Chain.* Retrieved May 25, 2024 from https://www.sonatype.com/state-of-the-software-supply-chain/introduction

[39] César Soto-Valero, Deepika Tiwari, Tim Toady, and Benoit Baudry. 2023. Automatic specialization of third-party java dependencies. *IEEE Transactions on Software Engineering* 49, 11 (2023), 5027–5045.

[40] Yuqiang Sun, Daoyuan Wu, Yue Xue, Hangbo Liu, Haijun Wang, Zhengzi Xu, Xiaofei Xie, and Yang Liu. 2024. GPTScan: Detecting Logic Vulnerabilities in Smart Contracts by Combining GPT with Program Analysis. *Proceedings of the 46th International Conference on Software Engineering* (2024), 2048–2060.

[41] Vasudev Vikram, Isabella Laybourn, Ao Li, Nicole Nair, Kelton OBrien, Rafaello Sanna, and Rohan Padhye. 2023. Guiding greybox fuzzing with mutation testing. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 929–941.

[42] Yao Wan, Wei Zhao, Hongyu Zhang, Yulei Sui, Guandong Xu, and Hai Jin. 2022. What do they capture? a structural analysis of pre-trained language models for source code. In *Proceedings of the 44th International Conference on Software Engineering*. 2377–2388.

[43] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2017. Skyfire: Data-driven seed generation for fuzzing. In *Proceedings of the 2017 IEEE Symposium on Security and Privacy*. 579–594.

[44] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2019. Superion: Grammar-aware greybox fuzzing. In *Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering*. 724–735.

[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 *Proceedings of the 2020 IEEE International Conference on Software Maintenance and Evolution*. 35–45.

[46] Yan Wang, Chao Zhang, Xiaobo Xiang, Zixuan Zhao, Wenjie Li, Xiaorui Gong, Bingchang Liu, Kaixiang Chen, and Wei Zou. 2018. Revery: From proof-of-concept to exploitable. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 1914–1927.

[47] wikipedia. 2024. *Cohen's kappa Wikipedia.* Retrieved May 29, 2024 from https://en.wikipedia.org/wiki/Cohen%27s_kappa

[48] Susheng Wu, Wenyan Song, Kaifeng Huang, Bihuan Chen, and Xin Peng. 2024. Identifying Affected Libraries and Their Ecosystems for Open Source Software Vulnerabilities. In *Proceedings of the 46th International Conference on Software Engineering*. 1–12.

[49] Susheng Wu, Ruisi Wang, Kaifeng Huang, Yiheng Cao, Wenyan Song, Zhuotong Zhou, Yiheng Huang, Bihuan Chen, and Xin Peng. 2024. Vision: Identifying Affected Library Versions for Open Source Software Vulnerabilities. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*.

[50] Yulun Wu, Zeliang Yu, Ming Wen, Qiang Li, Deqing Zou, and Hai Jin. 2023. Understanding the threats of upstream vulnerabilities to downstream projects in the maven ecosystem. In *Proceedings of the 2023 IEEE/ACM 45th International Conference on Software Engineering*. 1046–1058.

[51] Yingfei Xiong, Xinyuan Liu, Muhan Zeng, Lu Zhang, and Gang Huang. 2018. Identifying patch correctness in test-based program repair. In *Proceedings of the 40th international conference on software engineering*. 789–799.

[52] Congying Xu, Bihuan Chen, Chenhao Lu, Kaifeng Huang, Xin Peng, and Yang Liu. 2022. Tracking patches for open source software vulnerabilities. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 860–871.

[53] Meiqiu Xu, Ying Wang, Shing-Chi Cheung, Hai Yu, and Zhiliang Zhu. 2022. Insight: Exploring Cross-Ecosystem Vulnerability Impacts. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–13.

[54] Michal Zalewski. 2014. *American fuzzing lop (AFL).* Retrieved May 25, 2024 from http://lcamtuf.coredump.cx/afl/

[55] Zhengran Zeng, Hanzhuo Tan, Haotian Zhang, Jing Li, Yuqun Zhang, and Lingming Zhang. 2022. An extensive study on pre-trained models for program understanding and generation. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 39–51.

[56] 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 *Proceedings of the 2021 IEEE/ACM 43rd International Conference on Software Engineering*. 1695–1707.

[57] Lyuye Zhang, Chengwei Liu, Sen Chen, Zhengzi Xu, Lingling Fan, Lida Zhao, Yiran Zhang, and Yang Liu. 2023. Mitigating persistence of open-source vulnerabilities in maven ecosystem. In *Proceedings of the 2023 IEEE/ACM 38th International Conference on Automated Software Engineering*. 191–203.