Tabby: Automated Gadget Chain Detection for Java Deserialization Vulnerabilities

Xingchen Chen[1,2], Baizhu Wang[3], Ze Jin[1,2], Yun Feng[1,2], Xianglong Li[1,2], Xincheng Feng[4], Qixu Liu[1,2](✉)
[1] Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China
[2] School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China
[3] MYbank AntGroup, Hangzhou, China
[4] FG Security Lab AntGroup, Hangzhou, China
{chenxingchen, jinze, fengyun, lixianglong, liuqixu}@iie.ac.cn, wh1t3p1g@gmail.com, infosec_fxc@163.com

***Abstract*— Java is one of the preferred options of modern developers and has become increasingly more prominent with the prevalence of the open-source culture. Thanks to the serialization and deserialization features, Java programs have the flexibility to transmit object data between multiple components or systems, which significantly facilitates development. However, the features may also allow the attackers to construct gadget chains and lead to Java deserialization vulnerabilities. Due to the highly flexible and customizable nature of Java deserialization, finding an exploitable gadget chain is complicated and usually costs researchers a great deal of effort to confirm the vulnerability. To break such a dilemma, in this paper, we introduced *Tabby*, a highly accurate framework that leverages the Soot framework and Neo4j graph database for finding Java deserialization gadget chains. We leveraged *Tabby* to analyze 248 Jar files, found 80 practical gadget chains, and received 7 CVE-IDs from Xstream and Apache Dubbo. They both improved the security design to deal with potential security risks.**

***Index Terms*—code property graph, program analysis, vulnerability detection**

I. Introduction

Java is one of the most widely used programming languages today, owing to its convenience and cross-platform capabilities. According to Statista's 2022 report [1], 33.27% of developers use Java for their programming needs. However, the features that make Java so popular also make it vulnerable to security risks, particularly Java deserialization vulnerabilities (JDVs). In recent years, a number of high-risk and severe-impact JDVs have been identified, such as the Apache Log4j 2 remote code execution vulnerability (CVE-2021-44228) [2]. This vulnerability allows attackers to construct a malicious JNDI injection [3] to execute malicious commands on a remote server by returning a serialized payload. The Cyberint research team found that millions of servers were affected by this vulnerability [4].

Java deserialization vulnerability. Serialization is a process of converting the state of an object into a byte stream, while deserialization is the reverse process that recreates the actual Java object in memory by calling a specific method (e.g., `readObject`). However, if an application performs deserialization on untrusted data without sufficiently verifying that the resulting data is valid, an attacker can exploit this vulnerability by passing deliberately crafted data, causing the program to create unintended objects. The creation of such objects with their `readObject` method-call stack may lead to serious problems, such as arbitrary code execution. Such a vulnerability is called a *Java deserialization vulnerability (JDV) and is considered one of the most high-risk and covert Java vulnerabilities. Due to the flexibility of Java deserialization (can execute from any class that overrides methods such as `readObject`), potential ways to trigger vulnerabilities are hidden in a large number of possible conditions, making it challenging for researchers to identify them.* (see §II-A).

(✉): Corresponding author

Why finding JDVs is difficult. To understand why Java deserialization vulnerabilities are difficult to audit, we conducted an in-depth root cause analysis and identified the following main root causes. (**a**) For vulnerabilities such as SQL injection [5] and cross-site scripting (XSS) [6], we can determine whether the user's input is filtered **before** it is passed to the dangerous method (e.g., `statement.execute()`) or fuzz the parameters in the HTTP requests. However, for deserialization vulnerabilities, we cannot rely solely on input analysis. Instead, we need to focus on the data flow **after** the input data is passed into the source method. Specifically, we need to determine whether the method-call stack starting from the source method (which is typically the beginning of a gadget chain such as `object.readObject()` and `object.readExternal()`, usually overridden by developers of dependency libraries) can be executed to the sink method (which is typically the end of a gadget chain, as listed in Table VII). Because many classes contain source or sink methods, it is difficult to determine the control flow that can execute from the source method to a sink method unless the class objects are constructed. (**b**) Exploiting deserialization vulnerabilities involves serializing an object or even instantiating other class objects to assign property values of the object. During deserialization when exploiting JDVs, the control flow will eventually be executed to call a sink method (e.g., `Runtime.exec()`) when executing the logic of the serialized-object source method (e.g., `object.readObject()`) to call some methods of its class properties. The method-call stack from the source method to the sink method is known as a **gadget chain**, which requires clever combinations of many class objects. Moreover, the polymorphic nature [7] of Java (e.g., subclasses) leads to a large number of potential gadget chains. Existing state-of-the-art analysis tools (e.g., Spotbugs [8], CodeQL [9]) can only detect JDVs at the source code level and lack the ability to analyze libraries (dependency jar files, which are typically numerous and widespread in actual projects). However, JDVs can potentially exist in large numbers with the help of libraries, and existing tools are inadequate for finding JDVs.

Thus, this work's **main challenge** is to overcome the two root causes above and find exact gadget chains that can lead to vulnerability not only from source code, also from libraries (dependency jar files).

In addition, researchers often face difficulties in manually tracking gadget chains or customizing existing tools, as the latter either do not offer support for customized analysis or require repetitive analysis of the source code for every new customization. Moreover, fine-grained adjustments are often necessary for custom analysis, which becomes time-consuming and error-prone when the source code is re-analyzed for every customization. Thus, a significant **technical challenge** lies in creating intermediate data structures that can store all potential data and facilitate multiple custom re-analyses. Our solution to this challenge is to construct an intermediate data structure using code property graph.

To this end, we propose *Tabby*, a control flow analysis framework that utilizes a novel static analysis technique to automatically detect a gadget chain that leads to Java deserialization vulnerability. Based on our understanding of Java static analysis techniques and JDV, and inspired by detection approach [10], *Tabby* leverages our innovative scheme to build code property graph (see § III-B2). Then, we use the state-of-the-art Neo4j graph databases to store the complete semantic information generated by *Tabby* (see § III-B). As all the semantic information we need is stored in the graphs, *Tabby* leverages the high-performance queries of the Neo4j graph database to search for truly available gadget chains. With the help of our plugin *tabby-path-finder*, we found 80 effective gadget chains and reported deserialization vulnerabilities to multiple popular frameworks.

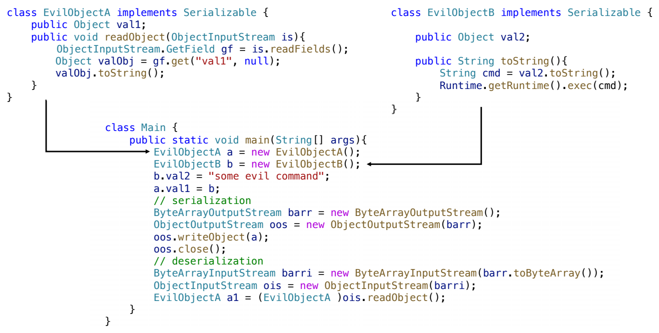
Contributions. The contributions are outlined as follows:

• We bring a new structure based on the code property graph, which can store all the potential information of the source code (potentially for improved analysis) and accelerate fine-grained Java program analysis.

• We proposed a new insight and algorithm to overcome the challenge of searching for an accurate gadget chain especially in Java libraries to trigger the Java deserialization vulnerability.

• We provide *Tabby*, a new framework to search Java deserialization vulnerabilities automatically.

• We conducted three experiments on 27 real-world projects, in which *Tabby* outperformed the state-of-the-art tools over 60.1% in accuracy.



```
class EvilObjectA implements Serializable {                    class EvilObjectB implements Serializable {
    public Object val1;
    public void readObject(ObjectInputStream is){                   public Object val2;
        ObjectInputStream.GetField gf = is.readFields();
        Object valObj = gf.get("val1", null);                       public String toString(){
        valObj.toString();                                              String cmd = val2.toString();
    }                                                                   Runtime.getRuntime().exec(cmd);
}                                                                   }
                                                                }
        class Main {
            public static void main(String[] args){
                EvilObjectA a = new EvilObjectA();
                EvilObjectB b = new EvilObjectB();
                b.val2 = "some evil command";
                a.val1 = b;
                // serialization
                ByteArrayOutputStream barr = new ByteArrayOutputStream();
                ObjectOutputStream oos = new ObjectOutputStream(barr);
                oos.writeObject(a);
                oos.close();
                // deserialization
                ByteArrayInputStream barri = new ByteArrayInputStream(barr.toByteArray());
                ObjectInputStream ois = new ObjectInputStream(barri);
                EvilObjectA a1 = (EvilObjectA )ois.readObject();
            }
        }
```

Fig. 1. An example of Java deserialization vulnerability

• We leveraged *Tabby* on popular Java open-source projects and found 80 gadget chains. Also, we got 7 CVE-IDs till now, and led the advanced and popular frameworks, Xstream [11] and Apache Dubbo [12], to improve their security design.

II. Background

A. Java deserialization vulnerability

Java supports a variety of serialization and deserialization mechanisms to ensure the consistency of object data between different Java Virtual Machines (JVMs). The serialization mechanism can transform the current object data in a JVM memory for persistence and save it as text data or binary data [13]. The receiving JVM can use the corresponding deserialization mechanism to revert the serialized object data to memory. Java applications can share data across devices and platforms through serialization and deserialization mechanisms. This mechanism is often used in Remote Procedure Call (RPC) and data passing during the interaction of various Web applications.

However, unverified serialized data can cause serious deserialization vulnerabilities for the receiving end of serialized data. Different deserialization mechanisms actively call certain specific methods. For example, the Java-native deserialization mechanism automatically calls the `readObject` method of the class. An attacker can make the receiver execute an unexpected program flow by passing in serialized data from a maliciously constructed class object.

Figure 1 shows a simple Java deserialization gadget chain, which can achieve the effect of arbitrary command execution. The example code uses the Java-native serialization and deserialization mechanism, which means the class object implements the `Serializable` interface. During the deserialization process, the `readObject` method of `EvilObjectA` reverts the class property `val1` and calls the `toString` method of `val1`. Additionally, the `toString` method of `EvilObjectB` calls the command execution method, while the specific command to be executed is the return value of `val2.toString()`. If we set the class property *val1* of `EvilObjectA` to `EvilObjectB`, then the `toString` of `EvilObjectB` is called automatically during the deserialization process. Finally, if the class property `val2` of

TABLE I
THE DETAILS OF GADGET CHAINS OF THE EXAMPLE

(source)EvilObjectA.readObject()
ObjectInputStream.readFields()
ObjectInputStream.get()
valObj.toString()
EvilObjectB.toString()
(sink)Runtime.getRuntime.exec()

`EvilObjectB` is set to a malicious command, the malicious command will be executed automatically during the deserialization process.

In a deserialization vulnerability, ***the method call stack*** *that can cause an exploit effect is called a* ***gadget chain***. In a gadget chain, the source methods are various methods that have a deserialization effect, and the sink methods are methods that can cause code execution or command injection. In the gadget chain of the example vulnerability (see Table I), the source method is `EvilObjectA.readObject()` and the sink method is `Runtime.getRuntime.exec()`. The gadget chains are commonly audited from the project's dependency libraries.

B. Code Property Graph and Graph Database

Code property graph (CPG) are graph data that contain multiple program semantic information. Many static program analysis approaches are based on control flow graphs and methods call graphs. When using static analysis to detect vulnerabilities, researchers typically perform semantic extraction of the code first and then identify vulnerabilities in the parsed semantic information. Most of the contents stored in memory during this process are class information, control flow graphs, method call graphs, etc. For example, for the method call graphs, the method body can be used as a data node to store the semantic information of itself, including the method name, method parameter list, method return value type, and other information. Moreover, a call relationship edge can exist between two method nodes, indicating a method call relationship between two methods.

For security researchers, traditional static analysis has a fixed logic for finding vulnerabilities and directly provides analysis results. The semantic information extracted during the analysis cannot be reused. When there are relevant features of vulnerabilities in the analysis results, security researchers cannot continue to analyze and identify them in an automated way, which is a waste of the previous static analysis results.

The emergence of graph databases provides a new option for the storage of static program analysis. A graph database [14] is a type of database that uses nodes, edges, and properties to represent and store data. Neo4j [15] (one of the most popular graph databases), supports a rich graph language query syntax to accomplish data retrieval. The introduction of graph databases splits the static program analysis process into two parts: semantic information extraction and specific vulnerability pattern identification. The extracted semantic information and preliminary analysis results are stored in the graph database. Researchers can re-use the graph database query syntax for vulnerability identification. In other words, security researchers can perform heuristic searches based on the results of previous queries.

III. TABBY DESIGN AND IMPLEMENTATION

A. Overview

In this section, we provide more details on accurately identifying gadget chains and explain the design and implementation of *Tabby*.

As we mentioned in § I, our work's **technical challenge** is to create intermediate data structures that store all potential data. To overcome this, we propose a novel data structure based on Code Property Graph (CPG), which involves converting Java code into graph nodes and edges and storing the results in Neo4j. As Figure 2 shows, we use the CPG in the Neo4j database to automate the search process for gadget chains by leveraging a novel algorithm we propose in § III-D.

Additionally, to address the **main challenge** of finding gadget chains in libraries (as discussed in § I), we developed a controllability analysis algorithm (detailed in §III-C) to help construct the CPG. This algorithm automates the process of analyzing variable controllability by simulating the actions of researchers and significantly saves time and effort in manual analysis. In addition, after the CPG is constructed successfully, security researchers can use the *tabby-path-finder* (see § III-D) with the static analysis results stored in the Neo4j graph database to customize their searches.

We present *Tabby* in three technical aspects: CPG Construction, Controllability Analysis, and Gadget Chain Identification. Firstly, we propose a high-level and novel scheme for constructing a code property graph (CPG) for object-oriented languages. This scheme extracts information on classes, methods, and more, and abstracts project codes into a CPG. Secondly, we introduce a controllability analysis algorithm that prunes the method call graph into a precise call graph, aiding in the construction of the CPG in the first aspect. This algorithm also calculates a property that can speed up the static analysis process. Finally, in the third aspect, we propose a Depth-First algorithm to identify gadget chains on the CPG.

B. Code Property Graph Construction

The Code Property Graph $G = (V, E)$ is a directed graph that consists of two sets of nodes: the `Class` data nodes (V_c) and the `Method` data nodes (V_m), and a set of relationship edges E that are a subset of $(V \times V)$ and include the following types of edges: `Extend`, `Interface`, `Has`, `Alias`, and `Call`. Each edge e in E is a two-tuple $\langle v_i, v_j \rangle$ where $v_i \in V$ and $v_j \in V$. These five edges represent the relationships between V_c and V_m. The correspondence between edges and nodes is shown in Table II, where v_{ci} represents a class node and v_{mi} represents a method node. To construct the Code Property Graph, several graphs are merged together, including the Object Relationship Graph (ORG), Method Call Graphs (MCG), and Method Alias Graphs (MAG). As shown in the bottom graph of Figure 4, the CPG is created by combining

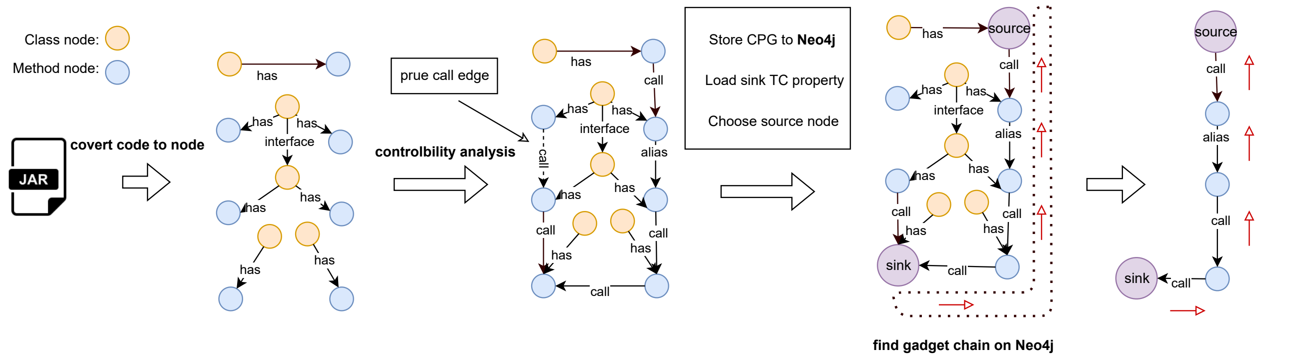
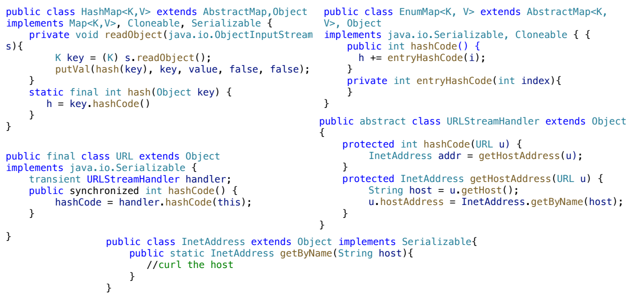


Fig. 2. Overall Workflow of Tabby



```
public class HashMap<K,V> extends AbstractMap,Object
implements Map<K,V>, Cloneable, Serializable {
    private void readObject(java.io.ObjectInputStream
s){
         K key = (K) s.readObject();
         putVal(hash(key), key, value, false, false);
    }
    static final int hash(Object key) {
       h = key.hashCode()
    }
}

public final class URL extends Object
implements java.io.Serializable {
    transient URLStreamHandler handler;
    public synchronized int hashCode() {
         hashCode = handler.hashCode(this);
    }

}
```

```
public class EnumMap<K, V> extends AbstractMap<K,
V>, Object
implements java.io.Serializable, Cloneable { {
    public int hashCode() {
      h += entryHashCode(i);
    }
    private int entryHashCode(int index){
    }
}

public abstract class URLStreamHandler extends Object
{
    protected int hashCode(URL u) {
         InetAddress addr = getHostAddress(u);
    }
    protected InetAddress getHostAddress(URL u) {
         String host = u.getHost();
         u.hostAddress = InetAddress.getByName(host);
    }
}
```

```
public class InetAddress extends Object implements Serializable{
    public static InetAddress getByName(String host){
       //curl the host
    }
}
```

Fig. 3. Core Code of the URLDNS chain

TABLE II
THE FIVE RELATIONSHIP EDGES

Relationship edges	Meaning
$E_{Interface}\langle v_{c1}, v_{c2}\rangle$	c1 $\rightarrow$ c2 , class c2 is an interface of class c2
$E_{Extend}\langle v_{c1}, v_{c2}\rangle$	c1 $\rightarrow$ c2, class c1 extends class c2
$E_{Has}\langle v_{c1}, v_{m1}\rangle$	c1 $\rightarrow$ m1, method m1 belongs to class c1
$E_{Call}\langle v_{m1}, v_{m2}\rangle$	m1 $\rightarrow$ m2, method m1 calls method m2
$E_{Alias}\langle v_{m1}, v_{m2}\rangle$	m1 $\rightarrow$ m2, The class of m2 is the superclass or interface of the class of m1

the ORG, PCG, and MAG. We used the semantic information extracted by Tabby to construct these three graphs.

1) Semantic Information Extraction: Tabby extracts semantic information from Java source code or Jar files as the initial step for conducting simple analysis. To achieve this, *Tabby* transforms a Java class into an intermediate code format that contains basic class information (such as ClassName and Modifier-Types), class method information (such as Method-Name, Modifier-Types, and Parameter Types), and class relationship information (such as Superclass and Interface). The extracted semantic information forms the foundation for building the code property graph used in our analysis. Also, as the underlying engine of *Tabby*, Soot ([16], [17]) generates a corresponding control flow graph for each method.

2) Code Property Graph Construction: Once the semantic information is extracted, Tabby utilizes it to construct a Code Property Graph (CPG). Prior to building the CPG, the three basic graphs (ORG, PCG, and MAG) must be constructed. We use the classic URLDNS gadget chain [18] as an example

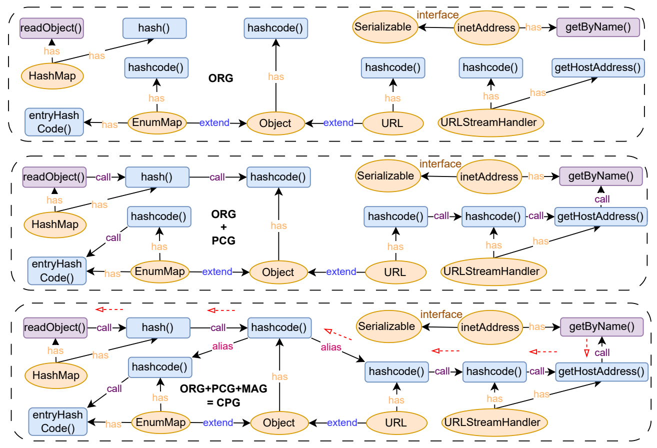


Fig. 4. Example of Code Property Graph

to better illustrate the construction process, and summarize the core code that triggers the vulnerability in Figure 3. The URLDNS gadget chain is triggered by the method-call stack `HashMap.reobject()`$\rightarrow$`HashMap.hash()`$\rightarrow$`URL.hashCode()`$\rightarrow$`URLStreamHandler.hashCode()`$\rightarrow$`URLStreamHandler.getHostAddress()`$\rightarrow$`inertAddress.getByName()`.

Object Relationship Graph Extraction. The Object Relationship Graph (ORG) serves as the foundation for constructing the code property graph, which include data nodes (V_c and V_m) for all classes that undergo analysis. Our design leverages the extracted semantic information to create class nodes and method nodes within ORG (see top graph of Figure 4) and establish relationship edges between them (as illustrated in the top three rows of Table II).

Precise Call Graph Extraction. To generate the Precise Call Graph (PCG), *Tabby* first constructs a Method Call Graph (MCG) that captures all external method calls within the current method body. This is achieved by traversing the control flow graph generated during the Semantic Information Extraction phase to identify and establish method call relationships within the current method. Once the MCG is constructed, *Tabby* applies a controllability analysis algorithm (see §III-C) to prune it down to a more precise representation. This is done by analyzing the controllability of variables and removing the edges (method calls) corresponding to the uncontrollable variables in the MCG. The resulting graph is the PCG (located at the middle from top to bottom of Figure 4), which captures a more accurate representation of the method calls in the code. During the process of building the PCG, our algorithm also records the method call details in the call edge properties.

Method Alias Graph Extraction. Polymorphism in Java allows for objects to be replaced by their subclasses or interface implementation classes, resulting in changes to the method execution flow. During static analysis, it can be challenging to determine which method call will trigger a deserialization vulnerability directly. The URLDNS gadget chain is triggered by `HashMap.hash()` calling the `hashCode()` method of any class (e.g., `URL.hashcode()`, `EnumMap.hashcode()`) since every class in Java is a subclass of `Object`. The potential replacement cases for `hashCode()` exist for the current class method, parent class method, and all implemented interface methods, each having the same method name, return value, and number of method parameters. To address this challenge, we design an Alias edge that connects the current method to the method in its parent class (see bottom graph of Figure 4). In this way, the potential gadget chains are connected on the graph. Combined with our design in § III-C and § III-D, the gadget chains can be found by traversing the code property graph. The extraction process is illustrated in Formula 1, where $e\langle v_{c1}, v_{c2}\rangle$ represents a relationship edge, v_{c1} and v_{c2} are class nodes, and v_{m1} and v_{m2} are method nodes.

$$\begin{aligned} f_{alias}(V_{m1}, V_{m2}) = \exists e\langle v_{c1}, v_{c2}\rangle \in E_{Extend} \cup E_{Interface} \\ (v_{c1} \in E_{has}\langle v_{c1}, v_{m1}\rangle, v_{c2} \in E_{has}\langle v_{c2}, v_{m2}\rangle) \end{aligned} \tag{1}$$

As illustrated in Figure 4, we establish connections between all the methods that can substitute the `hashcode()` method of the `Object` class using the Alias edge, which enables us to link the gadget chain from `readObject()` to `getByName()` in the graph. However, since the method of the Alias edge may not ultimately invoke the sink method (such as `EnumMap.hashcode()` $\rightarrow$ `EnumMap.entryHashCode()`), we developed the *tabby-path-finder* plugin (see § III-D) for the Neo4j database to locate the gadget chain by searching upwards from the sink method, thereby avoiding the calculation of such situations.

Code Property Graph Construction. We establish the relationships between nodes by constructing three fundamental graphs: ORG, PCG, and MAG. Once these graphs are built, we interconnect the class and method nodes extracted from the semantic information to form the code property graph.

C. Variable Controllability Analysis

In the previous section (see § III-B2), we explained how we use a control flow analysis algorithm to precisely prune the Method Call Graph (MCG) into a Precise Call Graph (PCG) by removing the edges corresponding to uncontrollable variables in the MCG. In this section, we will provide further details on this variable controllability analysis algorithm, which is crucial as it constitutes the core algorithm we employ to address the main challenge of detecting gadget chains as mentioned in § I. We put significant effort into pruning the MCG to accurately analyze and capture the behavior of method calls in the code. To ensure an efficient analysis process, we leverage intermediate data of methods generated during the analysis to speed it up (path explosion problems often arise in program analysis).

Essentially, we determine the origin of variables by traversing the control flow graph within the method body. As the algorithm traverses the method call statement, it abstracts the origin of the variable involved in the method as a property of the call edge. Additionally, we preserve the details of the method calls through controllability analysis, which is useful for identifying gadget chains (see § III-D). This enables us to utilize the properties on the call edges to extrapolate the method calls in actual cases when searching for gadget chains (as discussed in § III-D).

Design idea. Our controllability algorithm employs a field-sensitive points-to analysis algorithm [19], [20] to identify uncontrollable method calls, i.e., those not affected by deserialization. It focuses on analyzing the controllability of a collection of variables within a method. For intraprocedural analysis (within the method), we infer the possible values of variables in the method body. For interprocedural analysis (method calls), we infer the controllability of the method's return value and any potential changes in the controllability of the method parameters within the method body. Notably, the interprocedural analysis yields more accurate results for variable controllability. Without it, when a variable is passed into another method, it's unknown whether its value changes or not, leading many existing tools [10], [21] to default to it not changing (still controllable), resulting in a high false positive rate.

Design details. We would like to provide a detailed explanation of the controllability analysis algorithm using examples. Let's consider Figure 5(a) as an example. When the program reaches the 5th line, the variable `a2` will point to the method parameter `a`, making it a controllable variable. However, the initial parameter `a` will be replaced by `new A()`, becoming an uncontrollable variable. After executing the 5th line (see Figure 5(a)), the `b` property of the `a` object will point to the parameter `b`, meaning that `a.b` will be a controllable variable, while the initial parameter `b` will be replaced by `new B()` and become an uncontrollable variable. Finally, the example method will return the `a2` variable (the content of the original method parameter `a`), making it a controllable variable.

To represent the changes in method parameters and return values within the method body, we designed the **Action** property of method nodes, which is defined as follows. We explain the meaning of the variables in the Action property in Table III. In Figure 5(b), we represented the method parameters `a` (`param-1`), `b` (`param-2`), and the return value

TABLE III
MEANING OF VARIABLES IN ACTION

Variables	Meaning
this	class that the method belongs to
this.x	class property x of the class that the method belongs to
final-param-i	the final status of the initial method parameter i
final-param-i.x	class property x of final method parameter i
init-param-i	the initial status of method parameter i
init-param-i.x	class property x of initial method parameter i
return	return value
null	uncontrollable

(`return`) as an Action property.

- **Action**: A $\langle key, value\rangle$ pair array property of method nodes represents the origins of method parameters and return values after a method call, where *key* $\in$ {this, this.x, final-param-i, final-param-i.x, return}, and *value* $\in$ {this, this.x, init-param-j, init-param-j.y, null}.

The pseudocode of the analysis algorithm is shown in Algorithm 1. We performed our controllabilitiy analysis for all the methods in the MCG (Method Call Graphs).

The `doMethodAnalysis` function analyzes the Control Flow Graph (CFG) that was generated during the semantic information extraction phase (see § III-B1) for the method being analyzed. Using the `doAssignStmtAnalysis` function, it determines the controllability of variables within the method and saves the results as an Action property. The `doAssignStmtAnalysis` function analyzes all 15 statements, which contain semantic information. It assigns variable controllability weights to the **localMap** based on the rules defined in Table IV.

Algorithm 1 Controllability Points-to Analysis

```
Require: Jimple Control flow graphs(CFG)
Ensure: The change in controllability of input variables
1: function $doMethodAnalysis(CFG)$
2:     Initialize the result set results
3:     Initialize the variable set localMap
4:     for $Jimple\ statement$ (stmt) in $CFG$ do
5:         if stmt $is\ a\ return\ statement$ then
6:             $calculate\ the\ Action\ from$ localMap;
7:             $Add\ the\ current\ result\ set\ to$ results;
8:         else if stmt $is\ a\ method\ call\ statement$ then
9:             $extract\ the\ CFG_s\ of\ the$ stmt;
10:            if stmt $is\ a\ controllable\ method\ call$ then
11:                $Establishing\ call\ edge\ and\ PP$;
12:                Action $= doMethodAnalysis(CFG_s)$;
13:                out $= calc(Action, in)$;
14:                localMap $= correct(localMap, out)$;
15:            end if
16:        else
17:            $doAssignStmtAnalysis$(stmt, localMap);
18:        end if
19:    end for
20:    return results
21: end function
```

TABLE IV
VARIABLE CONTROLLABLE TRANSMISSION/ELIMINATION RULES

Type	Stmt	Rules
Original assignment	a = b;	b$\rightarrow$a
Create a new variable	a = new statement;	destroy the original CA[1] of a
Class property assignment	a.f = b;	b $\rightarrow$ a.f
Class property loading	a = b.f;	b.f $\rightarrow$ a
Static property assignment	Class.field = b;	b $\rightarrow$ Class.field
Static property loading	a = Class.field;	Class.field $\rightarrow$ a
Array assignment	a[i] = b;	b $\rightarrow$ a[i]
Array loading	a = b[i];	b[i] $\rightarrow$ a
Forced type conversion	a = (T) b;	b $\rightarrow$ a
Function return	return stmt;	return the CA[1] of stmt
method call assignment	a = b.func(c);	the return value of b.func $\rightarrow$ a
method call	b.func(c);	The content of func may change the CA[1] of b and c

[1]**CA**: controllability

TABLE V
CONTROLLABLE WEIGHTING RULES

Controllable weighting	Meaning
∞	Current variable is not controllable
0	Current variable comes from the caller class or class property
[1,n]	Current variable comes from the list of method parameters

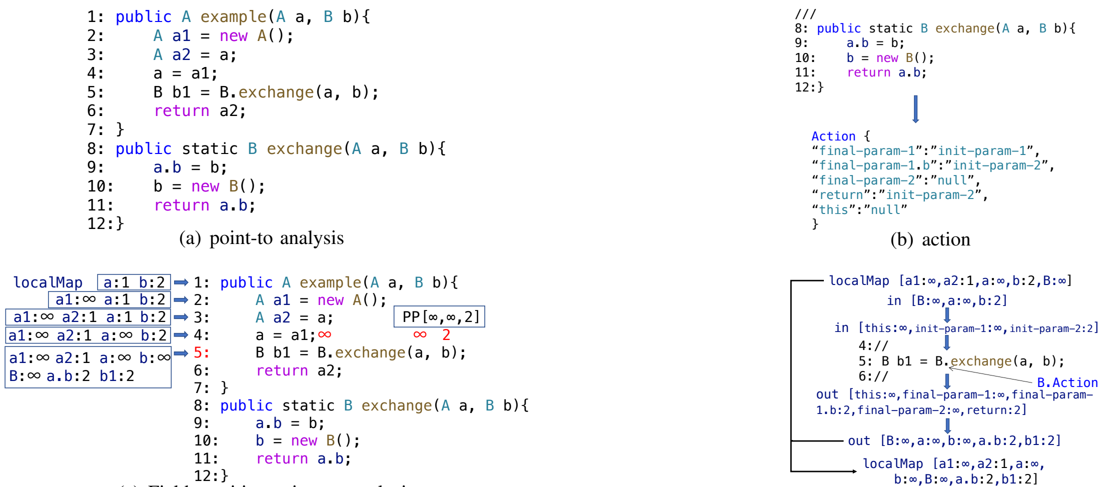
We used the weighting rules in Table V to express the controllability of the variables and designed the **P**olluted_**P**osition (PP) property of the CALL relationship edge.

- **Polluted_Position**: An array property of CALL relationship edge that represents the controllability weighting of method callers and method arguments.

The `doMethodAnalysis` function extracts the **P**olluted **P**osition (PP) from the **localMap** (as outlined in Algorithm 1) for subsequent taint analysis (see §III-D) when analyzing method call statements (which correspond to the last two rows in Table IV). To illustrate, in Figure 5(c), the `doAssignStmtAnalysis` assigns variable controllability weights for each line of the source code, resulting in a PP of [∞,∞,2] for the method `example` – CALL $\rightarrow$ `exchange` (i.e., the 5th line in Figure 5(b)). Additionally, Figure 5(c) shows how the values in **localMap** (as described in Algorithm 1) change.

To calculate the change in method parameters (`in` in Figure 5(d)), we utilized Formula 2 and obtained the output (`out` in Figure 5(d)) via the Action property. The Action property also serves as a caching mechanism: when encountering a previously analyzed method call during the analysis process, the algorithm will utilize the cached Action property of the method to speed up the analysis.

$$f_{calc}(Action, in) = \{\langle x, z\rangle | \langle x, y\rangle \in Action, \exists\langle y, z\rangle \in in\} \quad (2)$$



```
1: public A example(A a, B b){
2:     A a1 = new A();
3:     A a2 = a;
4:     a = a1;
5:     B b1 = B.exchange(a, b);
6:     return a2;
7: }
8: public static B exchange(A a, B b){
9:     a.b = b;
10:    b = new B();
11:    return a.b;
12:}
```

(a) point-to analysis

```
///
8: public static B exchange(A a, B b){
9:     a.b = b;
10:    b = new B();
11:    return a.b;
12:}

Action {
"final-param-1":"init-param-1",
"final-param-1.b":"init-param-2",
"final-param-2":"null",
"return":"init-param-2",
"this":"null"
}
```

(b) action

```
localMap  a:1 b:2                   1: public A example(A a, B b){
          a1:∞ a:1 b:2              2:     A a1 = new A();
          a1:∞ a2:1 a:1 b:2         3:     A a2 = a;          PP[∞,∞,2]
          a1:∞ a2:1 a:∞ b:2         4:     a = a1;∞            ∞  2
          a1:∞ a2:1 a:∞ b:∞         5:     B b1 = B.exchange(a, b);
          B:∞ a.b:2 b1:2            6:     return a2;
                                    7: }
                                    8: public static B exchange(A a, B b){
                                    9:     a.b = b;
                                    10:    b = new B();
                                    11:    return a.b;
                                    12:}
```

(c) Field-sensitive points-to analysis

```
localMap [a1:∞,a2:1,a:∞,b:2,B:∞]
      in [B:∞,a:∞,b:2]
in [this:∞,init-param-1:∞,init-param-2:2]
   4://
   5: B b1 = B.exchange(a, b);
   6://                                   B.Action
out [this:∞,final-param-1:∞,final-param-
1.b:2,final-param-2:∞,return:2]
out [B:∞,a:∞,b:∞,a.b:2,b1:2]
localMap [a1:∞,a2:1,a:∞,
     b:∞,B:∞,a.b:2,b1:2]
```

(d) calc and correct

Fig. 5. Controllability Analysis

TABLE VI
TRIGGER WEIGHTING RULES

Trigger weighting	Meaning
0	the method caller needs to be controllable
[1,n]	Which parameter needs to be controllable

As the logic of the callee method may change the variable controllability in the caller, we correct the localMap (see Figure 5(d)) by `out` (see Figure 5(d)) via Formula 3.

$$f_{correct}(localMap, out) = out \cup \{\langle x, y\rangle | \langle x, y\rangle \in localMap, \forall \langle p, q\rangle \in out, x \neq p\} \tag{3}$$

The `doMethodAnalysis` function prunes CALL edges when all values in their PP property are ∞, which helps to skip the analysis of uncontrollable method calls. This pruning of uncontrollable method calls helps to alleviate the path explosion problem in the algorithm. Additionally, the PP property can efficiently assist in generating gadget chains for gadget chain finding (as described in §III-D).

As all the information required for finding gadget chains is already stored in the CPG, the subsequent search for gadget chains involves querying the information in the CPG. We explain the query algorithm in the following section.

D. Gadget Chains Finding

In the preceding section (refer to §III-C), we elucidated on how the controllability analysis algorithm enriches Call relationship edges with method call particulars, i.e., **P**olluted_**P**osition (PP). In this section, we will delve into how to leverage a gadget chain finding algorithm to discover gadget chains via emulating actual method calls from sink methods utilizing PP and relationship edges. Our design philosophy is that if the algorithm can successfully traverse the source node as per the sink method's set initial conditions, then the path followed constitutes a gadget chain.

TABLE VII
PART OF THE SINK METHODS

Method	Type	TC
java.nio.file.Files.newOutputStream()	FILE	[1]
java.io.File.delete()	FILE	[0]
java.lang.reflect.Method.invoke()	CODE	[0,1]
java.net.ClassLoader.loadClass()	CODE	[0,1]
javax.naming.Context.lookup()	JNDI	[1]
java.rmi.registry.Registry.lookup()	JNDI	[1]
java.lang.Runtime.exec()	EXEC	[1]
java.lang.ProcessImpl.start()	EXEC	[1]
javax.xml.parsers.DocumentBuilder.parse()	XXE	[1]
javax.xml.transform.Transformer.transform()	XXE	[1]
java.net.InetAddress.getByName()	SSRF	[1]
java.net.URL.openConnection()	SSRF	[0]
java.lang.Object.readObject()	JDV	[0]

[1] TC is the **Trigger_Condition** that we defined in §III-D

As a prerequisite for gadget chains finding, we summarized 38 sink methods, some of which are in Table VII (the complete table is available on our website [22]). We tagged the **T**rigger_**C**ondition (TC) property of the sink methods and represented its weighting rules in Table VI, which is defined as follows. We summarize all TCs for each sink method in Table VII and use TC and PP in the following algorithm to determine whether CALL edges can be utilized.

- **Trigger_Condition**: An array property of the sink method represents which parameters must be controllable to achieve the attack effect.

$$f_{Traverse}(TC, PP) = \{PP[x] | x \in TC\} \tag{4}$$

The gadget chains finding is based on the CALL and ALIAS relationship edges in the CPG and is divided into `Expander` and `Evaluator`. The `Expander` determines whether the PP of the relationship edge meets the requirements of TC, and the `Evaluator` determines whether the search depth exceeds the setting or the target node is reached. The pseudocodes of `Expander` and `Evaluator` are shown in Algorithms 2 and 3. In Algorithm 2, we passed the sink's TC through the PP of the call relationship edge via Formula 4. The `Traverse` calculates the TC_{next} based on the edge's PP and the current TC. When encountering an ALIAS relationship edge, we passed the TC to TC_{next} directly. In Algorithm 3, we determined whether the last node of the path is a source node. If not, we would determine if the $getdepth(path)$ is less than the `depth` we set.

Algorithm 2 Expander

```
Require: path, $node_{end}$, relationship, TC
Ensure: $path_{next}$, $TC_{next}$
 1: function $Expander(path, node_{end}, relationship, TC)$
 2:     $node_{new} = getOtherNode(relationship, node_{end})$
 3:     if $RelationshipType\ is\ CALL$ then
 4:         $TC_{next} = f_{Traverse}(TC, CALL.PP)$
 5:         if $\exists x \in TC_{next}, x = \infty$ then
 6:             return $path, TC$
 7:         end if
 8:     end if
 9:     if $RelationshipType\ is\ ALIAS$ then
10:         $TC_{next} = TC$
11:     end if
12:     $path_{next} = path + node_{new}$
13:     return $next\_path, TC_{next}$
14: end function
```

Algorithm 3 Evaluator

```
Require: source, path, depth
Ensure: true or false
 1: function $Evaluator(source, path, depth)$
 2:     $node_{end} = getEndNode(path)$
 3:     if $node_{end} = source$ then
 4:         return $gadget\ chain$
 5:     else if $getdepth(path) < depth$ then
 6:         return $true$
 7:     else
 8:         return $false$
 9:     end if
10: end function
```

An example of gadget chains finding is shown in Figure 6. The method nodes (`A`, `C`, `C1`, `C2`, `E`, `G`, `H`, `I`, `E`, `E1`, `J`) in Figure 6 are connected by CALL or ALIAS relationship edges. Note that we denote the sink method as `A`, and the source method as `H`. We excludes `E`, `I` by `Expander`, and `G` by `Evaluator`. Taking the gadget chain `I`–CALL→`C1`–ALIAS→`C`–CALL→`A` as an example, one of the values in the `A`'s TC becomes ∞ when it passes through I–CALL→C1, indicating that it becomes uncontrollable during the passing process. We developed the Neo4j plugin *tabby-path-finder* for finding gadget chains, which is open-sourced on our website [22].

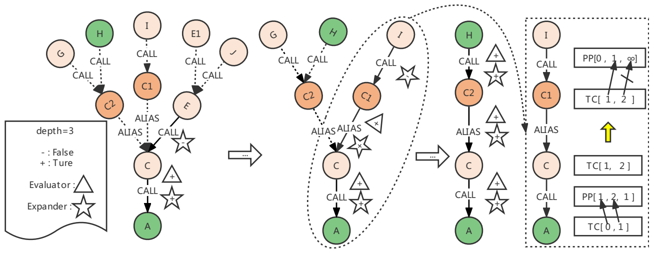
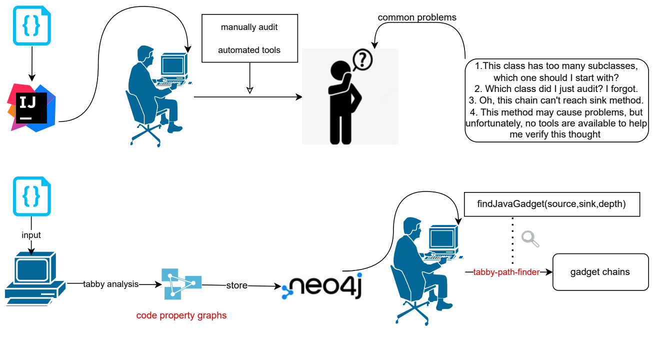


Fig. 6. An example of gadget chain finding



1.This class has too many subclasses, which one should I start with?
2. Which class did I just audit? I forgot.
3. Oh, this chain can't reach sink method.
4. This method may cause problems, but unfortunately, no tools are available to help me verify this thought

findJavaGadget(source,sink,depth)

Fig. 7. Comparison of gadget chain finding processes

As shown in Figure 7, we proposed a new solution for gadget chain finding. We abstracted the project code into code property graph and saved it in Neo4j. Then we could use *tabby-path-finder* to perform gadget chain finding. Security researchers can check for the existence of a gadget chain between any source and sink according to their needs, freeing them from the hassles of manual auditing and existing tools.

IV. Evaluation

A. Overview

To illustrate the effectiveness of the approach proposed in § III, we designed three main experiments in this section, including a test on the efficiency of code property graph generation, an existing gadget chain detection, and a gadget chain detection experiment on the development environment. The detailed results of the three experiments are published online on our website [22]. In the experiments, we aim to answer the following four research questions:

RQ1: What is the time-consuming construction process of code property graph?

RQ2: Compared to related works, how accurately is *Tabby* finding gadget chains in Java programs?

RQ3: How is *Tabby*'s performance on gadget chain detection in the real-world development environment?

RQ4: What can *Tabby* do for security researchers in defending against deserialization vulnerabilities?

B. Code Property Graph Generation Efficiency Experiment (RQ1)

TABLE VIII
EXPERIMENTAL DATA UNIT INFORMATION

Code amount (MB)	Jar file count	Class node count	Method node count	Relationship Edge count	Time consuming (min)
10	29	9055	59508	189021	1.9
20	63	14765	107623	341111	3.1
30	88	21104	153653	491651	6.0
40	93	25532	198130	628392	9.8
50	95	30859	249545	816421	12.7
100	113	32713	268670	857881	20.1
150	155	66247	503358	1587266	36.3

In this experiment, we used *Tabby* to extract semantic information from jar files selected from the top-100-popular jar files [23] and generate the corresponding code property graph. We recorded the time taken to generate the CPG for each generation (by repeating the code property graph generation 10 times for each experiment). To ensure that the data generated by the experiment are close to the actual generation efficiency, we removed the maximum and minimum time values obtained from the experiment and calculated the average value. The experiment results are shown in Table VIII.

As Table VIII shows, we observed an approximately linear correlation between the execution time and the count of class/method. In other words, given an increased amount of code, *Tabby* is unlikely to take unpredictable time to generate code property graph.

C. Comparison with State-Of-The-Art Tools (RQ2)

Many well-known projects (e.g. Apache Shiro [24], JBOSS [25]) have been confirmed to have deserialization vulnerabilities due to the existence of unauthorized deserialization interface and known gadget chains in the project's dependency libraries. Therefore, this experiment performs Java deserialization gadget chain detection on Java components with known gadget chains. We conducted experiments on all gadget chains of third-party dependency components in *ysoserial* [26] and *marshalsec* [27]. For each gadget chain, we used *Tabby* to build a code property graph and searched for gadget chains on Neo4j based on the source and sink. In addition, we compared our results with those obtained using *Gadgetinspector* [10] and *Serianalyzer* [21] on the same components. *Gadgetinspector* was presented at BlackHat 2018 to automatically detect Java deserialization gadget chains. *Serianalyzer* is an open-source tool proposed earlier than *gadgetinspector* for detecting gadget chains. We demonstrate the effectiveness of the solution proposed in this paper by comparing the result count, false positive rate, and false negative rate of the three tools. We would like to note that due to the high number of invalid gadget chains generated by *Serianalyzer* (often in the hundreds per component), we filter out those chains that do not contain the package name of the component being analyzed, using only the remaining chains as legitimate output.

We manually instantiated the classes in the three tools' gadget chains and wrote a Proof of Concept (PoC) to verify their effectiveness. We open-sourced all the PoCs online on our website [22]. If the program could execute from source to sink method after the deserialization construction, then the gadget chain is effective. The detailed results of the experiments are presented in Table IX. The column "Known in dataset" indicates the number of gadget chains related to these components in *ysoserial* and *marshalsec*. The "Result count" column in Table IX represents all chains output by the corresponding tool, including Fake gadget chains, Known gadget chains, and Unknown gadget chains. The "Known" column indicates how many of the gadget chains output by the tool are the same as those in the *ysoserial* and *marshalsec*. The "Unknown" column indicates effective gadget chains that are not recorded by *ysoserial* and *marshalsec*.

$$FPR = \frac{Fake\ gadget\ chain\ count}{result\ count} * 100\% \qquad (5)$$

$$FNR = \frac{Known\ in\ dataset\ - Knonw\ gadget\ chain\ count}{Known\ in\ dataset} * 100\% (6)$$

We calculated the false positive rate (FPR) of *Gadgetinspector*, *Serianalyzer* and *Tabby* according to Formula 5. The average FPR of *Tabby* was 32.9%, significantly lower than the 93.0% of *Gadgetinspector* and 98.6% of *Serianalyzer* (see Table IX). The fake gadget chains in *Tabby*'s output are mainly due to the conditional execution statements (i.e., **if-else**, **switch-case**). We calculated the false negative rate (FNR) of *gadgetinspector*, *Serianalyzer* and *Tabby* according to Formula 6. The average FNR of *Tabby* was 31.6%, significantly lower than the 86.8% of *gadgetinspector* and 81.6% of *Serianalyzer*. Furthermore, regarding unknown gadget chain detection, *Tabby* found a total of 27 new unknown gadget chains, including all unknown gadget chains found by *Gadgetinspector* and *Serianalyzer*. Some gadget chains in the dataset were not found by *Tabby* because they use the dynamic proxy [28], and we further discussed the limitations of *Tabby* in § V-B. Additionally, *Serianalyzer* was unable to output results for some components within an acceptable time (e.g., an hour), indicating that it may have had a problem with pruning during the call graph construction process.

Both *Gadgetinspector* and *Serianalyzer* utilized the ASM library to build call graphs and employed a data-flow algorithm for taint tracking. In contrast, *Tabby* conducts taint analysis on code property graph. Additionally, we elaborated on *Gadgetinspector* and *Serianalyzer* in § IV-F.

D. Development Environment Gadget Chain Detection (RQ3)

Java developers frequently use popular development frameworks or application deployment environments. In real-world application projects, the dependency jar files provided by these environments can be used by attackers to generate gadget chains. We chose three mainstream environments as target to evaluate *Tabby*'s effectiveness: Spring framework, JDK8, and a variety of development deployment middlewares. We analyzed all the dependency jar files from the three target frameworks. The experimental results are shown in Table X.

TABLE IX
THE EXPERIMENT RESULTS IN COMPARISON

Component	Known in dataset	Result count			Fake			Known			Unknown			FPR (%)			FNR (%)			time(min)		
		GI	TB	SL	GI	TB	SL	GI	TB	SL	GI	TB	SL	GI	TB	SL	GI	TB	SL	GI	TB	SL
AspectJWeaver	1	8	1	27	8	0	27	0	1	0	0	0	0	100	0	100	100	0	100	1.5	6.9	7.0
BeanShell1	1	2	3	1	2	2	1	0	1	0	0	0	0	100	66.7	100	100	0	100	1.5	6.5	5.6
C3P0	1	2	6	1	2	2	0	0	1	0	0	3	1	100	33.3	0	100	0	100	1.5	6.5	6
Click1	1	4	1	56	3	0	56	1	1	0	0	0	0	75	0	100	0	0	100	1.4	6.7	4.6
Clojure	1	12	2	X	9	1	X	1	1	X	2	0	X	75	50	X	0	0	X	1.3	7.3	X
CommonsBeanutils1	1	2	1	50	2	0	50	0	1	0	0	0	0	100	0	100	100	0	100	1.3	6.8	5.3
commons-colletions(3.2.1)	5	4	17	73	3	4	73	0	4	0	1	9	0	75	23.5	100	100	20	100	1.5	7.3	4.3
commons-colletions(4.0.0)	2	4	18	38	3	5	38	0	1	0	1	12	0	75	27.8	100	100	50	100	1.5	7.1	5.0
FileUpload1	2	3	2	6	2	0	4	1	2	2	0	0	0	66.7	0	33.3	50	0	0	1.3	7.2	5.0
Groovy1	1	4	2	137	4	2	137	0	0	0	0	0	0	100	100	100	100	100	100	1.7	7.1	7.3
Hibernate	2	2	4	55	2	0	55	0	2	0	0	2	0	100	0	100	100	0	100	1.6	7.3	7.0
JBossInterceptors1	1	2	3	7	2	2	6	0	1	1	0	0	0	100	66.7	85.7	100	0	0	1.4	6.6	5.3
JSON1	1	4	0	0	4	0	0	0	0	0	0	0	0	100	100	100	100	100	100	1.4	6.3	4.6
JavaassistWeld1	1	2	3	3	2	2	2	0	1	1	0	0	0	100	66.7	66.7	100	0	0	1.5	6.6	6.6
Jython1	1	42	2	X	42	2	X	0	0	X	0	0	X	100	100	X	100	100	X	1.5	7.8	X
MozillaRhino	2	3	1	93	3	0	93	0	1	0	0	0	0	100	0	100	100	50	100	1.6	7.0	4.0
Myface	1	2	1	0	2	0	0	0	1	0	0	0	0	100	0	0	100	0	100	1.4	7.1	5.6
Rome	1	2	2	19	2	0	18	0	1	1	0	1	0	100	0	94.7	100	0	0	1.6	6.4	4.0
Spring	2	2	2	4	2	2	4	0	0	0	0	0	0	100	100	100	100	100	100	1.3	6.4	4.6
Vaadin1	1	6	1	18	5	0	18	1	1	0	0	0	0	83.3	0	100	0	0	100	1.5	7.2	5.6
Wicket1	2	3	2	5	2	0	3	1	2	2	0	0	0	66.7	0	60	50	0	0	1.4	6.7	4.6
commons-configration	1	2	0	0	2	0	0	0	0	0	0	0	0	100	0	0	100	100	100	1.5	6.4	5.3
spring-beans	2	2	2	0	2	1	0	0	1	0	0	0	0	100	50	0	100	50	100	1.5	6.9	4.6
spring-aop	2	6	2	0	6	1	0	0	1	0	0	0	0	100	50	0	100	50	100	1.5	6.7	6.0
XBean	1	2	1	0	2	0	0	0	1	0	0	0	0	100	0	0	100	0	100	1.4	6.5	4.3
Resin	1	2	0	0	2	0	0	0	0	0	0	0	0	100	0	0	100	100	100	1.7	7.4	7.3
Total	38	129	79	593	120	26	585	5	26	7	4	27	1	93.0	32.9	98.6	86.8	31.6	81.6	-	-	-

[1]**GI**: gadgetinspector [2]**TB**: Tabby [3]**SL**: serianalyzer [4]**X**:The process is not terminated

TABLE X
THE EXPERIMENT RESULTS OF DEVELOPMENT SCENES

Scenes	Version	Jar file count	Code size(MB)	Result count	effective gadget chains	FPR	searching time(s)
Spring	2.4.3	66	25.5	10	7	30%	8.2
JDK8	8u242	19	102.2	13	10	23.1%	10.2
Tomcat	8.5.47	25	7.9	4	3	25%	3.6
Jetty	9.4.36	67	10.3	6	4	33.3%	4.1
Apache Dubbo	3.0.2	15	13.6	5	3	40%	5.5

TABLE XI
PART OF GADGET CHAINS FOUND IN THE SPRING FRAMEWORK

org.#.aop.target.LazyInitTargetSource.getTarget() org.#.jndi.support.SimpleJndiBeanFactory.getBean(String) org.#.jndi.JndiLocatorSupport.lookup() javax.naming.Context.lookup()
org.#.aop.target.PrototypeTargetSource.getTarget() org.#.jndi.support.SimpleJndiBeanFactory.getBean(String) org.#.jndi.JndiLocatorSupport.lookup() javax.naming.Context.lookup()

[1] #: springframework

1) Spring Framework Experiment: The Spring framework [29] is a widely used framework for Java Web applications; its derivatives Spring Boot, Spring Cloud, and other preconfigured Spring components are popular among developers. Thus, the current Spring framework scene performs deserialization gadget chain detection on all the 66 Jar files that the Spring framework relies on. *Tabby* finds a total of 7 effective deserialization gadget chains. Some of the experimental results are shown in Table XI.

We found several unknown or unpublished details of deserialization gadget chains lurking in the Spring framework scene environment. These gadget chains consist of the Spring-Aop [30], Spring-TX [31], Spring-core [32], and logback-core [33]. The third item in Table XI was given as CVE-2020-11619 [34] on April 7, 2020, its exploit has not been published thus far, and the gadget chain is still included in Spring-Aop. *Tabby* also found two new gadget chains in Spring-Aop (#1 and #2 in Table XI) similar to the CVE-2020-11619 [34], which will launch JNDI injections to achieve arbitrary code execution.

2) JDK8 Experiment: The popularity of JDK8 [35] is higher among all versions of Java. Thus, the JDK8 experiment detects the deserialization gadget chains for all the 19 Jar files on which JDK8 depends. We found ten effective gadget chains, five of which bypass the deserialization blacklist of the XStream component [11] (Xstream is an open-source deserialization framework that allows users to conveniently perform serialization and deserialization operations). The gadget chains are shown in our website [22].

Among the five XStream gadget chains discovered by *Tabby*, one chain is a known gadget chain, and the other four are unknown gadget chains. We reported these 4 unknown gadget chains to the XStream team and received the corresponding CVE-IDs: CVE-2021-21346 [36], CVE-2021-21351 [37], CVE-2021-39147 [38] and CVE-2021-39152 [39].

3) Middleware Experiment: The middleware environment is for a fixed deployment environment of Java Web applications. We chose Tomcat [40], Jetty [41], and Apache Dubbo [12] for this experiment. *Tabby* found ten effective deserialization gadget chains, including five unknown chains. The gadget chains are shown in our website [22].

The main sink methods in the gadget chains detected in this experiment are `lookup`, `getConnection`, and `invoke`. An attacker can construct these gadget chains to execute arbitrary code. We reported 3 unknown gadget chains of Apache Dubbo [12] and received the corresponding CVE-IDs: CVE-2021-43297 [42], CVE-2022-39198 [43] and CVE-2023-23638 [44].

E. Result Description (RQ4)

In the two detection experiments conducted, *Tabby* was able to detect a total of 117 deserialization gadget chains (sum of the results in Tables IX and X for the respective experiments), out of which 80 gadget chains were found to be effective (categorized as either "known" or "unknown" in Table IX and the "effective gadget chains" column in Table X). The overall false positive rate for *Tabby* was observed to be 31.6%. The main reason for these false positives is attributed to certain logical judgments in the code, which disrupt the execution flow designed by the gadget chain.

These results indicate that the number of deserialization gadget chains lurking in third-party open-source components remains significant, implying that the current open-source environment may not be as secure as initially perceived.

Some well-known application (e.g., Xstream, Apache Dubbo) developers have blocked (blacklisted) the classes used in common gadget chains (e.g., the gadget chains in *ysoserial* [26]) and forbade the application from deserializing these classes. Security researchers in these teams can use *Tabby* to find potential gadget chains in their projects and refine the blacklist with classes from the gadget chains. Xstream and Apache Dubbo refined their blacklists based on the gadget chains we submitted and modified some details of their deserialization mechanism. We published all the gadget chains from our experiments on our website [22] and hope to help defend against deserialization vulnerabilities.

F. Reflections on Existing Tools

Upon careful consideration, we have reflected on the potential advantages of our tool over existing ones and identified possible areas for improvement in the latter. We believe that this analysis will provide valuable insights into the development and future research of vulnerability detection tools.

- The handling of Java polymorphism in Gadgetinspector seems to be incomplete, resulting in a less comprehensive call graph being constructed.
- Gadgetinspector has a feature that skips nodes that have already been traversed when searching for gadget chains, which helps reduce running costs but may also lead to the loss of potential chains.
- Both Gadgetinspector and Serianalyzer may experience a loss of precision in interprocedural analysis, but *Tabby* was designed from the outset with interprocedural analysis in mind.
- Unlike Gadgetinspector and Serianalyzer, Tabby retains all intermediate results (such as method calls) generated during the analysis process into Neo4j and provides query plugins. This allows users to query multiple times to verify their ideas, which is particularly useful for identifying defects.

V. Discussion

A. Overview

In this section, we discuss the limitations of *Tabby* and potential future improvements. We examine its limitations with the Dynamic Proxy feature of Java, the lack of capability to automatically confirm gadget chains with payloads that lead to false-positive chains, and its scalability to other object-oriented languages such as C# and PHP. Additionally, we discuss strategies to avoid vulnerabilities for the security community and project owners.

B. Dynamic Proxy and Reflection Limitation

Tabby is a control flow analysis approach to detect gadget chains and is adapted to most deserialization vulnerabilities. However, it is difficult to perform control flow analysis for finding gadget chains incorporated with reflection [45] (a Java feature that allows a running Java program to obtain information about itself and modify the properties of class objects) and dynamic proxy [28] (a use case for reflection that can create instances of the interface during runtime). The flexibility of dynamic proxy and reflection makes it difficult for static analysis to determine which class or method in the code will be used.

C. Automatically Confirming Gadget Chains with Payloads Limitation

Currently, *Tabby* cannot automatically generate malicious input payloads based on the identified gadget chains to confirm that the chains can definitely be triggered. In the future, we expect to leverage javassist [46] to generate payloads, combined with Java dynamic hooking technology using JVM Tool Interface (JVMTI) and Java Instrumentation API [47] to automatically check whether the gadget chain is correct.

D. Extending Our Approach to Other Object-Oriented Languages

Our code property graph construction approach is specifically designed for Java. However, our work can be generalized to other object-oriented languages, such as C# and PHP. Researchers can adapt our approach to utilize a static analysis engine to perform project gadget chain finding in these languages based on the generated code property graph. This opens up possibilities for future work to extend our approach beyond Java, and we believe it could be a promising direction for further research.

E. Strategies for Community to Avoid the JDVs

While Java Deserialization Vulnerabilities (JDVs) may potentially exist in a large number of projects, there are several potential strategies that could be employed to mitigate Java Deserialization Vulnerabilities (JDVs) in software projects. One approach is to implement blacklist/whitelist checks for deserialization, which can help prevent certain types of attacks. While using a whitelist is generally considered more effective than a blacklist, it requires developers to have knowledge of the potential deserialization risks in order to properly configure the list. Another potential mitigation strategy is to add token validation for the deserialization interfaces of the project, which can provide an additional layer of access control. These strategies, along with other best practices for secure coding, such as input validation and output encoding, can help reduce the risk of JDVs and improve the overall security of software projects.

VI. Related Work

A. Static Analysis for Java Vulnerability Detection

Static analysis directly extracts the semantic information of the program from the source code or compiled files for program security analysis. Its key techniques include data flow analysis, interprocedural analysis, symbolic execution, etc. Livshits et al. [48] used static program analysis techniques to detect security vulnerabilities in Java applications, such as SQL injection, XSS, etc. Following studies [49]–[52] improved the effectiveness of static detection for specific exploits of vulnerabilities. Some studies extended the static analysis to detect vulnerabilities in DOS [53], Logic vulnerabilities [54], etc. Hammer et al. [55] proposed IFC (information flow control) based on PDGs (program dependence graphs) to find security vulnerabilities in programs. Tripp et al. [56] proposed taint analysis to generate useful answers in a limited time and space and implemented the TAJ for static taint analysis in Java. Some later studies [57]–[64] can detect multiple types of vulnerabilities. Static analysis is normally accompanied by a high false positive rate, and some studies [65], [66] have made attempts to reduce the false positive rate. Industrial Java static analysis tools such as SpotBugs (Fingbugs) [8], [62], Fortify SCA [67], Checkstyle [68] and PMD [69] are widely used by security researchers to find bugs or vulnerabilities. However, none of these tools support finding gadget chains.

B. Code Property Graph

Several approaches have been proposed for detecting vulnerabilities using CPG (code property graph). Michael Martin et al. [70] proposed a program query language to find errors and security flaws in Java applications. Fabian Yamaguchi et al. [71] used CPG to model common vulnerabilities in the Linux kernel and found 18 unknown security vulnerabilities. Michael Backes et al. [72] extends [71] by adding method call graphs to make CPG more suitable for web languages. SGL (Security Graph Language) [73] enabled large-scale analysis of open source code graph structure datasets and vulnerability classification. Wang et al. [74] used a deep learning approach to improve the accuracy of previous defect analysis methods based on CPG. However, these approaches cannot analyze the class and method information needed for deserialization vulnerabilities. In 2019 Microsoft announced CodeQL [9], a CPG-based analysis tool that enabled the conversion of specified programming languages into CPG and vulnerability detection using user-defined graph database languages. CodeQL currently only supports converting source code to CPG. However, most third-party open-source components in Java deserialization are Jar file types. Moreover, CodeQL does not provide a methodology to find gadget chains directly. In our work, we modeled the deserialization vulnerability using CPG. As we already showed, we have addressed several designs and implementation challenges to enable *Tabby* to detect gadget chains automatically.

C. Java Deserialization Vulnerability

The FoxGlove Security team [75] revealed a real-world example of the Java deserialization vulnerability principle and the implementation of remote command execution with the underlying class library of Apache Commons Collections. Moreover, the vulnerability affected several well-known Java Web middleware, such as Weblogic, WebSphere, and JBoss. Peles et al. [76] found multiple SDKs with deserialization vulnerabilities under the Android platform that could lead to kernel-level arbitrary code execution. The NCC Group [77] detailed the principle and impact of the Java deserialization vulnerabilities. Currently, the leading Java deserialization vulnerability exploitation tools are ysoserial [26] and marshalsec [27], which summarize the typical java deserialization gadget chains. ObjectMap [78] is a scanner tool that uses publicly available gadget chains to fuzz deserialization vulnerabilities in web applications for Java and PHP. Still, this tool cannot find unknown gadget chains. Philipp Holzinger et al. [79] conducted an in-depth study of various Java vulnerabilities and highlighted deserialization as a risky feature because it may allow attackers to access objects they would not usually have access to. Ian Haken published his research on the automatic detection of Java deserialization gadget chains at Black Hat 2018 and open-sourced his development of gadgetinspector [10]. Shawn Rasheed et al. proposed a hybrid analysis system for detecting Java deserialization vulnerabilities [80]. Sayar et al. conducted an in-depth study of the triggering causes and fixes for java deserialization vulnerabilities but did not propose a methodology for automating the search for gadgets [81]. Wu et al. [82] used the method of passthrough call graph analysis to find the gadget chains, and did a rough comparison experiment with gadgetinspector in the two components of *ysoserial*, but they did not conduct a more detailed evaluation of the approach.

VII. Conclusion

We designed and implemented *Tabby*, an automated tool for detecting Java deserialization vulnerabilities. *Tabby*is highly effective and efficient. We leveraged *Tabby* to find 80 effective gadget chains on popular open-source projects and seven real vulnerabilities on popular frameworks. Our research overcame the challenges of finding gadget chains in Java projects, and will contribute to elevating the security assurance of the modern Java open-source community.

Acknowledgment

Xingchen Chen, Ze Jin, Yun Feng, Xianglong Li and Qixu Liu are supported in part by the Youth Innovation Promotion Association CAS (No.2019163), the Strategic Priority Research Program of Chinese Academy of Sciences (No. XDC02040100), the Key Laboratory of Network Assessment Technology at Chinese Academy of Sciences and Beijing Key Laboratory of Network security and Protection Technology.

REFERENCES

[1] "Worldwide developer survey most used languages," https://www.statista.com/statistics/793628/worldwide%2ddeveloper%2dsurvey%2dmost%2dused%2dlanguages, Accessed: April 2023.

[2] "CVE-2021-44228," https://nvd.nist.gov/vuln/detail/CVE-2021-44228, Accessed: April 2023.

[3] "BlackHat 2016," https://www.blackhat.com/docs/us-16/materials/us%2d16%2dMunoz%2dA%2dJourney%2dFrom%2dJNDI%2dLDAP%2dManipulation%2dTo%2dRCE%2dwp.pdf, Accessed: April 2023.

[4] "Cyberint: Log4J2 Remote Code Execution," https://cyberint.com/blog/research/cve%2d2021%2d44228%2dlog4j2%2drce/, Accessed: April 2023.

[5] "SQL Injection," https://owasp.org/www-community/attacks/SQL_Injection, Accessed: April 2023.

[6] "Cross Site Scripting," https://owasp.org/www-community/attacks/xss/, Accessed: April 2023.

[7] "Polymorphism in Java," https://docs.oracle.com/javase/tutorial/java/IandI/polymorphism.html, Accessed: April 2023.

[8] "SpotBugs," https://spotbugs.github.io/, Accessed: April 2023.

[9] "Codeql," https://github.com/github/codeql, Accessed: April 2023.

[10] "gadgetinspector," https://github.com/JackOfMostTrades/gadgetinspector, Accessed: April 2023.

[11] "Xstream," https://x-stream.github.io/, Accessed: April 2023.

[12] "Apache Dubbo," https://dubbo.apache.org/en/index.html, Accessed: April 2023.

[13] D. Poo, D. Kiong, and S. Ashok, "Object serialization and remote method invocation," in *Object-Oriented Programming and Java*. Springer, 2008, pp. 279–295.

[14] R. Angles and C. Gutierrez, "Survey of graph database models," *ACM Computing Surveys (CSUR)*, vol. 40, no. 1, pp. 1–39, 2008.

[15] "The Neo4j Graph Platform," https://neo4j.com/, Accessed: April 2023.

[16] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot: A java bytecode optimization framework," in *CASCON First Decade High Impact Papers*, 2010, pp. 214–224.

[17] "Soot - A framework for analyzing and transforming Java and Android applications," http://soot-oss.github.io/soot/, Accessed: April 2023.

[18] "Triggering a DNS lookup using Java Deserialization," https://blog.paranoidsoftware.com/triggering-a-dns-lookup-using-java-deserialization/, Accessed: April 2023.

[19] D. J. Pearce, P. H. Kelly, and C. Hankin, "Efficient field-sensitive pointer analysis of c," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 30, no. 1, pp. 4–es, 2007.

[20] B. Steensgaard, "Points-to analysis in almost linear time," in *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1996, pp. 32–41.

[21] "serianalyzer," https://github.com/mbechler/serianalyzer, Accessed: April 2023.

[22] "Supporting website for Tabby," https://sites.google.com/view/tabby-tool/home, Accessed: April 2023.

[23] "Top-100-JAVA-Libraries," https://jarcasting.com/top-100-java-libraries/, Accessed: April 2023.

[24] "CVE-2019-12422 of Apache Shiro," https://nvd.nist.gov/vuln/detail/CVE-2019-12422, Accessed: April 2023.

[25] "CVE-2017-7504 of JBOSS," https://nvd.nist.gov/vuln/detail/CVE-2017-7504, Accessed: April 2023.

[26] "ysoserial," https://github.com/frohoff/ysoserial, Accessed: April 2023.

[27] "marshalsec," https://github.com/mbechler/marshalsec, Accessed: April 2023.

[28] "Java Dynamic Proxy," https://docs.oracle.com/javase/8/docs/technotes/guides/reflection/proxy.html, Accessed: April 2023.

[29] "Spring," https://spring.io/, Accessed: April 2023.

[30] "Spring-Aop," https://javadoc.io/doc/org.springframework/spring-aop/latest/index.html, Accessed: April 2023.

[31] "Spring-Tx," https://javadoc.io/doc/org.springframework/spring-tx/latest/index.html, Accessed: April 2023.

[32] "Spring-Core," https://javadoc.io/doc/org.springframework/spring-core/latest/index.html, Accessed: April 2023.

[33] "logback-core," https://logback.qos.ch/, Accessed: April 2023.

[34] "CVE-2020-11619," https://nvd.nist.gov/vuln/detail/CVE-2020-11619, Accessed: April 2023.

[35] "Jdk8," https://www.oracle.com/java/technologies/javase/javase8-archive-downloads.html, Accessed: April 2023.

[36] "CVE-2021-21346," https://x-stream.github.io/CVE-2021-21346.html, Accessed: April 2023.

[37] "CVE-2021-21351," https://x-stream.github.io/CVE-2021-21351.html, Accessed: April 2023.

[38] "CVE-2021-39147," https://x-stream.github.io/CVE-2021-39147.html, Accessed: April 2023.

[39] "CVE-2021-39152," https://x-stream.github.io/CVE-2021-39152.html, Accessed: April 2023.

[40] "Tomcat," https://tomcat.apache.org/, Accessed: April 2023.

[41] "Jetty," https://www.eclipse.org/jetty/, Accessed: April 2023.

[42] "CVE-2021-43297," https://lists.apache.org/thread/1mszxrvp90y01xob56yp002939c7hlww, Accessed: April 2023.

[43] "CVE-2022-39198," https://lists.apache.org/thread/8d3zqrkoy4jh8dy37j4rd7g9jodzlvkk, Accessed: April 2023.

[44] "CVE-2023-23638," https://lists.apache.org/thread/8h6zscfzj482z512d2v5ft63hdhzm0cb, Accessed: April 2023.

[45] "Using Java Reflection," https://www.oracle.com/technical-resources/articles/java/javareflection.html, Accessed: April 2023.

[46] "Javassist," https://www.javassist.org/, Accessed: April 2023.

[47] "Java Instrument API," https://docs.oracle.com/javase/7/docs/api/java/lang/instrument/Instrumentation.html, Accessed: April 2023.

[48] V. B. Livshits and M. S. Lam, "Finding security vulnerabilities in java applications with static analysis." in *USENIX security symposium*, vol. 14, 2005, pp. 18–18.

[49] W. Huang, Y. Dong, and A. Milanova, "Type-based taint analysis for java web applications," in *International Conference on Fundamental Approaches to Software Engineering*. Springer, 2014, pp. 140–154.

[50] S. Thomas, L. Williams, and T. Xie, "On automated prepared statement generation to remove sql injection vulnerabilities," *Information and Software Technology*, vol. 51, no. 3, pp. 589–598, 2009.

[51] W. G. Halfond and A. Orso, "Amnesia: analysis and monitoring for neutralizing sql-injection attacks," in *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, 2005, pp. 174–183.

[52] L. K. Shar and H. B. K. Tan, "Automated removal of cross site scripting vulnerabilities in web applications," *Information and Software Technology*, vol. 54, no. 5, pp. 467–478, 2012.

[53] V. Wüstholz, O. Olivo, M. J. Heule, and I. Dillig, "Static detection of dos vulnerabilities in programs that use regular expressions," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2017, pp. 3–20.

[54] V. Felmetsger, L. Cavedon, C. Kruegel, and G. Vigna, "Toward automated detection of logic vulnerabilities in web applications," in *19th USENIX Security Symposium (USENIX Security 10)*, 2010.

[55] C. Hammer, M. Grimme, and J. Krinke, "Dynamic path conditions in dependence graphs," in *Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, 2006, pp. 58–67.

[56] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman, "Taj: effective taint analysis of web applications," *ACM Sigplan Notices*, vol. 44, no. 6, pp. 87–97, 2009.

[57] O. Tripp, M. Pistoia, P. Cousot, R. Cousot, and S. Guarnieri, "Andromeda: Accurate and scalable security analysis of web applications," in *International Conference on Fundamental Approaches to Software Engineering*. Springer, 2013, pp. 210–225.

[58] G. Zhao, H. Chen, and D. Wang, "Data-flow based analysis of java bytecode vulnerability," in *2008 The Ninth International Conference on Web-Age Information Management*. IEEE, 2008, pp. 647–653.

[59] J. Thomé, L. K. Shar, D. Bianculli, and L. C. Briand, "Joanaudit: A tool for auditing common injection vulnerabilities," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 1004–1008.

[60] J. Thomé, L. K. Shar, D. Bianculli, and L. Briand, "Security slicing for auditing common injection vulnerabilities," *Journal of Systems and Software*, vol. 137, pp. 766–783, 2018.

[61] M. Sridharan, S. Artzi, M. Pistoia, S. Guarnieri, O. Tripp, and R. Berg, "F4f: taint analysis of framework-based web applications," in *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, 2011, pp. 1053–1068.

[62] N. Ayewah, W. Pugh, D. Hovemeyer, J. D. Morgenthaler, and J. Penix, "Using static analysis to find bugs," *IEEE software*, vol. 25, no. 5, pp. 22–29, 2008.

[63] S. S. Alqahtani, E. E. Eghan, and J. Rilling, "Sv-af—a security vulnerability analysis framework," in *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2016, pp. 219–229.

[64] F. Spoto, E. Burato, M. D. Ernst, P. Ferrara, A. Lovato, D. Macedonio, and C. Spiridon, "Static identification of injection attacks in java," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 41, no. 3, pp. 1–58, 2019.

[65] J. Yang, L. Tan, J. Peyton, and K. A. Duer, "Towards better utilizing static application security testing," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 2019, pp. 51–60.

[66] R. Mangal, X. Zhang, A. V. Nori, and M. Naik, "A user-guided approach to program analysis," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 462–473.

[67] "Fortify SCA," https://www.microfocus.com/en-us/cyberres/application-security/static-code-analyzer, Accessed: April 2023.

[68] "Checkstyle," https://checkstyle.sourceforge.io/, Accessed: April 2023.

[69] "PMD," https://pmd.github.io/, Accessed: April 2023.

[70] M. Martin, B. Livshits, and M. S. Lam, "Finding application errors and security flaws using pql: a program query language," *Acm Sigplan Notices*, vol. 40, no. 10, pp. 365–383, 2005.

[71] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *2014 IEEE Symposium on Security and Privacy*. IEEE, 2014, pp. 590–604.

[72] M. Backes, K. Rieck, M. Skoruppa, B. Stock, and F. Yamaguchi, "Efficient and flexible discovery of php application vulnerabilities," in *2017 IEEE european symposium on security and privacy (EuroS&P)*. IEEE, 2017, pp. 334–349.

[73] D. Foo, M. Y. Ang, J. Yeo, and A. Sharma, "Sgl: A domain-specific language for large-scale analysis of open-source code," in *2018 IEEE Cybersecurity Development (SecDev)*. IEEE, 2018, pp. 61–68.

[74] W. Xiaomeng, Z. Tao, W. Runpu, X. Wei, and H. Changyu, "Cpgva: code property graph based vulnerability analysis by deep learning," in *2018 10th International Conference on Advanced Infocomm Technology (ICAIT)*. IEEE, 2018, pp. 184–188.

[75] "Foxglovesecurity," https://foxglovesecurity.com/2015/11/06/what%2ddo%2dweblogic%2dwebsphere%2djboss%2djenkins%2dopennms%2dand%2dyour%2dapplication%2dhave%2din%2dcommon%2dthis%2dvulnerability/, Accessed: April 2023.

[76] O. Peles and R. Hay, "One class to rule them all: 0-day deserialization vulnerabilities in android," in *9th USENIX workshop on offensive technologies (WOOT 15)*, 2015.

[77] R. C. Seacord, "Combating java deserialization vulnerabilities with look-ahead object input streams (laois)," *NCC Gr Whitepaper*, 2017.

[78] N. Koutroumpouchos, G. Lavdanis, E. Veroni, C. Ntantogian, and C. Xenakis, "Objectmap: Detecting insecure object deserialization," in *Proceedings of the 23rd Pan-Hellenic Conference on Informatics*, 2019, pp. 67–72.

[79] P. Holzinger, S. Triller, A. Bartel, and E. Bodden, "An in-depth study of more than ten years of java exploitation," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 779–790.

[80] S. Rasheed and J. Dietrich, "A hybrid analysis to detect java serialisation vulnerabilities," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 1209–1213.

[81] I. Sayar, A. Bartel, E. Bodden, and Y. Le Traon, "An in-depth study of java deserialization remote-code execution exploits and vulnerabilities," *ACM Transactions on Software Engineering and Methodology*, 2022.

[82] J. Wu, J. Zhao, and J. Fu, "A static method to discover deserialization gadget chains in java programs," in *Proceedings of the 2022 2nd International Conference on Control and Intelligent Robotics*, 2022, pp. 800–805.