Improving Java Deserialization Gadget Chain Mining via Overriding-Guided Object Generation

Sicong Cao[†*], Xiaobing Sun[†✉], Xiaoxue Wu[†✉], Lili Bo[†✉], Bin Li[†], Rongxin Wu[‡],
Wei Liu[†], Biao He[§], Yu Ouyang[§], Jiajia Li[§]
[†]Yangzhou University [‡]Xiamen University [§]Ant Group
[†]{DX120210088, xbsun, xiaoxuewu, lilibo, lb, weiliu}@yzu.edu.cn, [‡]wurongxin@xmu.edu.cn,
[§]{hb187361, yu.oyy, jiajia.lijj}@antgroup.com

***Abstract*—Java (de)serialization is prone to causing security-critical vulnerabilities that attackers can invoke existing methods (gadgets) on the application's classpath to construct a gadget chain to perform malicious behaviors. Several techniques have been proposed to statically identify suspicious gadget chains and dynamically generate injection objects for fuzzing. However, due to their incomplete support for dynamic program features (e.g., Java runtime polymorphism) and ineffective injection object generation for fuzzing, the existing techniques are still far from satisfactory.**

In this paper, we first performed an empirical study to investigate the characteristics of Java deserialization vulnerabilities based on our manually collected 86 publicly known gadget chains. The empirical results show that 1) Java deserialization gadgets are usually exploited by abusing runtime polymorphism, which enables attackers to reuse serializable overridden methods; and 2) attackers usually invoke exploitable overridden methods (gadgets) via dynamic binding to generate injection objects for gadget chain construction. Based on our empirical findings, we propose a novel gadget chain mining approach, *GCMiner*, which captures both explicit and implicit method calls to identify more gadget chains, and adopts an overriding-guided object generation approach to generate valid injection objects for fuzzing. The evaluation results show that *GCMiner* significantly outperforms the state-of-the-art techniques, and discovers 56 unique gadget chains that cannot be identified by the baseline approaches.

***Index Terms*—Java deserialization vulnerability, gadget chain, method overriding, exploit generation**

I. Introduction

Java serialization [1] enables an application to convert an object to a stream of bytes. By contrast, Java deserialization reconstructs the original object from its serialized byte stream. In spite of the convenience of Java serialization in cross-platform data transmission and persistence storage [2], deserializing data from untrusted provenance provides an entry point for diverse attacks, including denial of service (DoS) attacks [3], [4] and remote code execution (RCE) [5]. In such attacks, an attacker reuses exploitable code fragments (so called *gadgets*) on the application's classpath and joins them together piece by piece (so called *gadget chains*) to facilitate a malicious injection object flowing into the security-sensitive call site (e.g., `Method.invoke()`) [6]. A deserialization vulnerability disclosed recently, namely `Spring4Shell` [7], allows attackers to send queries to create web shells to servers running the `Spring` framework [8], leading to RCE. The impact of the `Spring4Shell` vulnerability can be devastating because 60% of developers use `Spring` for their Java applications development. Due to its severity, OWASP (Open Web Application Security Project) lists *Insecure Deserialization* as the top 10 most critical web application security risks in 2017 [9].

The root cause of deserialization vulnerabilities is that the control and data flow can be manipulated by attackers to enable the malicious deserialized objects to *reach* (in terms of control flow) and *affect* (in terms of data flow) the security-sensitive sink [6]. Some techniques [10], [11] have been proposed to automatically mine exploitable gadget chains. *Gadget Inspector* [10] performs static taint analysis [12], [13] to track inter-procedural data flows, and applies the Breadth-First Search (BFS) to identify gadget chains from deserialization entry points (sources) to security-sensitive call sites (sinks). Considering that such a purely static solution may suffer from precision issues and requires manual inspection of the reports, SerHybrid [11] adopts a hybrid analysis solution, which constructs the heap access paths to find source objects that affect security-sensitive call sites and utilizes fuzzing [14], [15] to generate actual injection objects, to verify whether the sinks are reachable.

Nevertheless, *SerHybrid* has limited effectiveness due to two reasons. First, *SerHybrid* performs points-to analysis [16] to identify source-to-sink method execution paths. However, due to the dynamic features (e.g., runtime polymorphism [17]) of Java language, any available overridden method (gadget) on the application's classpath may be exploited to construct gadget chains, resulting in high false negatives. Second, *SerHybrid* generates injection objects based on heap access paths for gadget chain verification. This heap access path reflects the taint propagation flow from a source object to the security-sensitive call site. However, due to the unawareness of hard constraints (requiring dynamically modifying the properties of an injection object to trigger the target gadget chain) introduced by certain gadgets, such generated injection objects may be semantically invalid. Hence, fuzzing solutions blind to the structure of a given gadget chain will get stuck in the initial fuzzing stage.

In this paper, we first performed an empirical study to investigate the characteristics of Java deserialization vulnerabilities. In particular, we focused on answering the following two ques-

[*]Work done during internship at Ant Group.
[✉]Corresponding authors.

1558-1225/23/$31.00 ©2023 IEEE
DOI 10.1109/ICSE48619.2023.00044

tions: 1) how Java deserialization gadgets are exploited; and 2) how gadget chains are constructed. We manually constructed a real-world Java deserialization vulnerability benchmark, which consists of the well-known *ysoserial* [18] repository and 18 popular Java applications. In total, 86 (52 out of which are new) publicly exploitable gadget chains are included in our benchmark. From the empirical results, we found that 1) Java deserialization gadgets are usually exploited by abusing advanced language features (e.g., runtime polymorphism), which enables attackers to reuse serializable overridden methods on the application's classpath; and 2) attackers usually invoke exploitable overridden methods (gadgets) via dynamic binding [19] to generate injection objects for gadget chain construction.

Based on our empirical findings, we propose a novel Gadget Chain Miner, *GCMiner*, which considers dynamic program features (Java runtime polymorphism) to identify more exploitable gadget chains and generates valid injection objects through dynamic binding for fuzzing. *GCMiner* performs static analysis to construct the *Deserialization-Aware Call Graph* (DA-CG) to model both explicit and implicit (method overriding) method calls to identify more gadget chains. To verify whether a statically identified gadget chain is exploitable, *GCMiner* adopts an overriding-guided object generation approach to generate exploitable injection objects for fuzzing. Using the overriding relations between methods as a guidance for injection object generation can effectively help the fuzzer to be aware of the object structures that reflect the target gadget chain. Gadget chains which receive an injection object reachable to their security-sensitive call sites will be outputted as exploitable chains.

To evaluate the effectiveness, we compared *GCMiner* with two state-of-the-art gadget chain mining tools, *Gadget Inspector* [10] and *Serhybrid* [11]. Our experimental results demonstrate that *GCMiner* significantly outperforms the baselines.

In summary, this paper makes the following contributions:

- **An empirical study:** We performed an empirical study on 86 exploitable gadget chains from several famous open-source Java projects. Our findings show that dynamic features of Java language are abused to construct gadget chains by generating injection objects through dynamically binding exploitable gadgets.
- **A novel gadget chain mining approach:** We propose a gadget chain mining approach that identifies more exploitable gadgets by considering deserialization-related dynamic features of Java language, and generates injection objects through dynamically binding overridden methods to verify statically reported gadget chains. We open-sourced *GCMiner* and the benchmark to facilitate further research[1].
- **Technique evaluation:** We performed comprehensive experiments to evaluate the effectiveness of *GCMiner*. The experimental results reveal that *GCMiner* discovers 56 unique gadget chains missed by state-of-the-art baselines.

[1]https://github.com/GCMiner/GCMiner

II. Background and Motivation

A. Terminology

In this section, we introduce several basic concepts which will be used in this paper.

Java deserialization vulnerability. A Java deserialization vulnerability is a security bug that can be exploited when the Java application deserializes untrusted data. Attackers could inject crafted objects into deserialization-related methods (e.g., `readObject()`), which pass malicious commands to a security-sensitive call site, resulting in diverse attacks like RCE.

Gadget and gadget chain. To facilitate an injection object reaching the security-sensitive call site, attackers should reuse a series of exploitable methods in memory to manipulate the deserialization process to achieve their desired malicious behaviors. Such a sequence of method calls is called a *gadget chain*, and each method of this chain is called a *gadget* [6]. The presence of a gadget chain on the application's classpath is one of the necessary conditions to carry out deserialization attacks.

Magic method and security-sensitive call site. The security-risk of a gadget chain comes from its first gadget/method (source) that can be invoked *automatically* during object deserialization. Such a self-executing method is called a *magic method*. These magic methods can be exploited by attackers to bypass existing security defenses to deserialize their crafted injection objects. Correspondingly, the last gadget/method (sink) which performs the malicious command carried by the injection object is called a *security-sensitive call site*.

Property-oriented programming. To exploit a Java deserialization vulnerability, attackers have to instantiate an object of attacker-controlled type and modify its properties to trigger the execution of an exploitable gadget chain. Such a technique used in constructing this injection object is called *Property-Oriented Programming* (POP) [20]. POP allows an attacker to manipulate the data- and control-flow of the victim application, thereby exploiting existing gadgets on the application's classpath for deserialization attacks.

B. Motivating Example

To better clarify the above terminology and illustrate the motivation of our approach, we take a real-world Java deserialization vulnerability (CVE-2021-21346[2]) in `XStream` [21] as an example. `XStream` is a popular Java deserialization library to serialize objects to XML and back again.

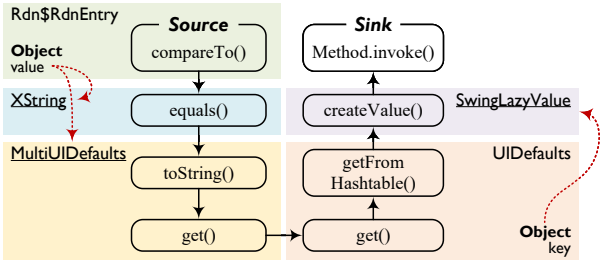
Figure 1a presents the simplified code snippet of the target gadget chain in CVE-2021-21346. The serializable class `javax.naming.ldap.Rdn` (line 1) invokes the magic method `compareTo()` (line 3) automatically during object deserialization. Such a magic method can be regarded as the *source* or *entry point* that allows attackers to inject malicious objects. If a program performs object deserialization without additional security check, an arbitrary method (e.g., `Runtime.exec(command)` [22]) specified in a string `className`

[2]https://x-stream.github.io/CVE-2021-21346.html

```
1  /*javax.naming.ldap.Rdn$RdnEntry.class*/
2  private Oject value;
3  public int compareTo(RdnEntry that) { /*Source or Magic Method*/
4    if (value.equals(that.value)) {...}
                  Overriding
5  /*com.sun.org.apache.xpath.internal.objects.XString.class*/
6  public boolean equals(Object obj2) { /*2nd gadget*/
7    return str().equals(obj2.toString()); }
                          Overriding
8  /*javax.swing.MultiUIDefaults.class*/
9  public synchronized String toString() { /*3rd gadget*/
10   Enumeration keys = keys();
11   while (keys.hasMoreElements()) {
12     Object key = keys.nextElement();
13     buf.append(key + "=" + get(key) + ","); ...}
14 public Object get(Object key) { /*4th gadget*/
15    Object value = super.get(key); ...}

16 /*javax.swing.UIDefaults.class*/
17 public Object get(Object key) { /*5th gadget*/
18   Object value = getFromHashtable(key); ...}
19 private Object getFromHashtable(final Object key) { /*6th gadget*/
20  if (value instanceof LazyValue) {
21    try {
22      value = ((LazyValue)value).createValue(this); ...}}
                        Overriding
23 /*sun.swing.SwingLazyValue.class*/
24 public Object createValue(final UIDefaults table) { /*7th gadget*/
25   try {
26     Class<?> c = class.forName(className, true, null);
27     if (methodName != null) {
28       Class[] types = getClassArray(args);
29       Method m = c.getMethod(methodName, types);
30       makeAccessible(m);
31       return m.invoke(c, args); /*Sink or Security-Sensitive Call Site*/
```

(a) A simplified code snippet with deserialization vulnerability.



(b) The stack trace of the gadget chain in Figure 1a.

Fig. 1: An example Java deserialization vulnerability and the gadget chain.

(line 26) carried by this malicious injection object will be invoked (line 31), leading to RCE.

To exploit this vulnerability, an attacker chains existing gadgets on the application's classpath to enable the malicious injection object to flow from the source to the sink. For example, to invoke the second gadget `equals()` at line 6, an attacker instantiates the class `Xstring` (line 5) and assigns this instance to the injection object `Rdn$RdnEntry`'s property `value` (line 2) by POP. This dynamic method call takes advantage of Java polymorphism. Similarly, the third gadget (`toString()` at line 9) and seventh gadget (`createValue()` at line 24) should also be dynamically invoked to facilitate the propagation of the injection object. Note that the fourth gadget (`get()` at line 14) to sixth gadget (`getFromHashtable` at line 19) can be directly invoked during deserialization. Figure 1b depicts the corresponding stack trace of the gadget chain. In general, constructing such an exploitable gadget chain requires 1) modifying the injection object's properties to trigger the execution of an exploitable gadget chain, and 2) assigning proper values to reach the security-sensitive call site.

[Observation-1] *Dynamic program features (e.g., Java runtime polymorphism) can be abused by attackers to implement insecure deserialization paths.*

As shown in Figure 1a, since class `XString` to which the gadget `equals()` (line 6) belongs inherits the superclass `Object`, the tainted property `value` of an injection object `RdnEntry` can continue to propagate through the call statement `value.equals()` (line 4). Taking such a dynamic runtime behavior into account can effectively identify attacker-controlled gadgets on the application's classpath. A straightforward solution is to build Call Graphs (CGs) [23] to perform interprocedural analysis, as existing gadget chain mining tools [10], [11] do. However, these statically constructed CGs either ignore these serialization-related dynamic features or handle them partially [17], [24], resulting in unsound results.

Therefore, for gadget chain identification, the consideration of dynamic program features may help avoid false negatives.

[Observation-2] *Constructing an exploitable gadget chain requires dynamically invoking a series of overridden methods to enable the tainted objects to pass into the dangerous sink.*

Due to the imprecision of static analysis [25], most statically reported gadget chains cannot be exploited in practice, resulting in a high *false-positive rate*. To alleviate this problem, generating exploitable injection objects to verify the exploitability of suspicious gadget chains is an effective solution [26]–[28]. However, generating injection objects that follow the execution trace of gadget chains to be verified is challenging because of the nested class hierarchy, i.e., the properties of an injection object need to be modified to reach the dangerous sink. For instance, to construct the deserialization gadget chain in Figure 1a, an attacker needs to invoke attacker-controlled overridden methods (e.g., `equals()` at line 6) on the application's classpath multiple times to implement a customized deserialization routine. This dynamic deserialization behavior can be realized through POP, and is implemented by determining the method to invoke at runtime (i.e., dynamic binding). In this motivating example, the purpose of the first two dynamic bindings is to invoke `XString.equals()` (line 6) and `MultiUIDefaults.toString()` (line 9) respectively to facilitate the propagation of the tainted property `value` (line 2), while the third is to ensure that the injection object can reach the security-sensitive call site `Method.invoke()` (line 31) to execute malicious commands. However, the heap access path adopted by *SerHybrid* fails to infer these implicit control flows, making it hard to reach dangerous sinks.

Therefore, for gadget chain verification, generating injection objects to trigger given gadget chains via dynamic binding may improve the effectiveness of fuzzing.

TABLE I: Benchmark information.

Library	Affected Application	#Chain	Type
-	ysoserial	34	-
YAML	JBoss RESTEasy	1	RCE
	Apache Camel	2	
	Apache Brooklyn	1	
	Apache XBean	1	
JDK	Shiro	3	JNDIi
	Pippo	2	RCE
BlazeDS	Adobe Coldfusion	2	RCE
	VMWare VCenter	1	
Red5	Red5	1	RCE
Hessian	Hessian	5	RCE
XStream	XStream	14	RCE SRA
Others	Commons Collections	3	RCE
	Dubbo	2	RCE
	WebLogic	5	RCE JNDIi
	Emissary	3	SSRF
	Jenkins	2	RCE
	Apache OFBiz	3	RCE
	Spring	1	JNDIi
Total		**86**	-

III. Empirical Study

A. Experiment Setup

Inspired by the above two observations, we performed an empirical study to investigate the characteristics of Java deserialization vulnerabilities. Particularly, we aim to answer the following two research questions:

- **RQ1:** How are Java deserialization gadgets exploited?
- **RQ2:** How are gadget chains constructed?

The answers to these questions provide empirical foundations on 1) whether dynamic program features are abused by attackers to implement insecure deserialization; and 2) whether dynamically invoking overridden methods on the application's classpath are widely exploited to construct gadget chains.

Data collection. We chose *ysoserial* [18] repository, a famous project that provides 34 Java payloads with corresponding gadget chains exploited in publicly known deserialization attacks, as part of our dataset. Considering the scarcity of gadget chains available for analysis, we *manually* collected public Java deserialization gadget chains from well-known vulnerability disclosure platforms such as National Vulnerability Database (NVD) [29], Common Vulnerabilities and Exposures (CVE) [30], Exploit Database (Exploit-DB) [31], etc. We selected target applications that satisfied the following criteria. First, they are Java open-source projects since the characteristics of deserialization vulnerabilities might vary among different programming languages [32]. Second, they support deserialization operations and have been reported to have the risk of being exploited so that gadget chains we mined are available. Third, they contain sufficient information (e.g., *Proof-of-Concept* (POC) and affected versions) to verify the authenticity of gadget chains.

Table I shows the details of the benchmark. Columns "Library" and "Affected Application" present the deserialization libraries that cause the vulnerabilities and corresponding affected applications. Note that due to the re-implementation of deserialization operations (i.e., not relying on any deserialization library), column "Library" of some applications is labeled as "Others". Column "#Chain" represents the number of collected gadget chains. Column "Type" presents different vulnerability types in each application, including *Remote Code Execution* (RCE), *JDNI Injection* (JNDIi), *System Resource Access* (SRA), and *Server-Side Request Forgery* (SSRF). In total, we collected 86 deserialization gadget chains covering 18 Java applications, 52 out of which are not included in *ysoserial* repository.

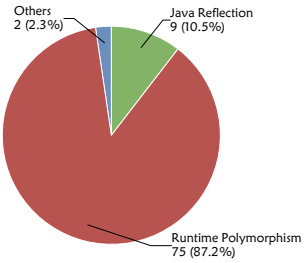


Fig. 2: Ways of exploiting available gadgets.

B. Exploitation of Java Deserialization Gadgets (RQ1)

An exploitable gadget chain requires: 1) a *magic method* (source or the first gadget) deserializing untrusted data that can be injected by attackers; 2) a *security-sensitive call site* (sink or the last gadget) that ultimately executes a dangerous operation; and 3) a series of *gadgets* facilitating the propagation of injection objects [33]. Hence, we first investigated exploitable magic methods and security-sensitive call sites in our benchmark. They are listed as follows.

- **Magic methods:** `hashCode`, `compareTo`, `toString`, `get`, `put`, `compare`, `readObject`, `readExternal`, `readResolve`, `finalize`, `equals`
- **Security-Sensitive Call Sites.**
  - ***Remote Code Execution*** **(RCE):** `getDeclaredMethod`, `getConstructor`, `exec`, `getMethod`, `loadClass`, `start`, `findClass`, `invoke`, `forName`, `newInstance`, `defineClass`, `<init>`, `exit`
  - ***JDNI Injection*** **(JNDIi):** `getConnection`, `connect`, `lookup`, `getObjectInstance`, `do_lookup`
  - ***System Resource Access*** **(SRA):** `newBufferedReader`, `newBufferedWriter`, `delete`, `newInputStream`, `newOutputStream`
  - ***Server-Side Request Forgery*** **(SSRF):** `openConnection`, `openStream`

In total, a set of 11 magic methods and 25 security-sensitive call sites are found in our benchmark. It is worth noting that only five magic methods and 11 security-sensitive call sites (highlighted in gray) are included in previous works [10], [11]. The direct consequence of missing these exploitable magic methods (e.g., `compareTo()` in Figure 1) and security-sensitive call sites is that some exploitable gadget chains cannot be identified.

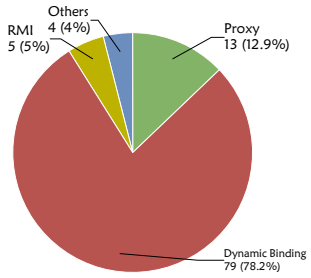


Fig. 3: Ways of gadget chain construction.

In addition, we further investigated how these gadgets were exploited. In particular, we focused on dynamic program features of Java language that may be abused by attackers. As shown in Figure 2, 75 out of 86 (87.2%) gadget chains exploit available gadgets by abusing Java runtime polymorphism to invoke overridden methods on the application's classpath. As described before, attackers can exploit these gadgets to pass malicious injection objects. In the remaining cases, only nine (10.5%) gadget chains rely on Java reflection to exploit gadgets.

[Finding-1] *Java deserialization gadgets are commonly exploited by abusing advanced language features (e.g., runtime polymorphism), which enables attackers to reuse serializable overridden methods on the application's classpath.*

C. Construction of Gadget Chains (RQ2)

Figure 3 shows the distribution of different ways for gadget chain construction. The results show that 79 out of 86 (91.9%) known gadget chains leverage dynamic binding (*at least* once) to modify the properties of injection objects to invoke overridden methods. Besides, in some cases, diverse ways (e.g., remote method invocation (RMI) [34] and dynamic proxy [35]) are mixed to trigger the execution of gadget chains[3]. For example, to exploit a known gadget chain `CommonsCollections1` [36] of *ysoserial*, attackers have to combine dynamic proxy and dynamic binding to invoke exploitable gadgets to reach the security-sensitive call site.

[Finding-2] *To construct exploitable gadget chains, attackers usually invoke exploitable overridden methods (gadgets) via dynamic binding to generate injection objects, which facilitate the malicious data flowing into dangerous sinks.*

IV. Methodology

Based on our empirical findings, we propose *GCMiner*, a novel gadget chain mining approach which takes dynamic program features (Java runtime polymorphism) into account to mine implicit method calls for gadget chain identification, and generates valid injection objects through dynamic binding for fuzzing.

[3]Note that a gadget chain constructed by multiple techniques will be repeatedly counted in Figure 3.

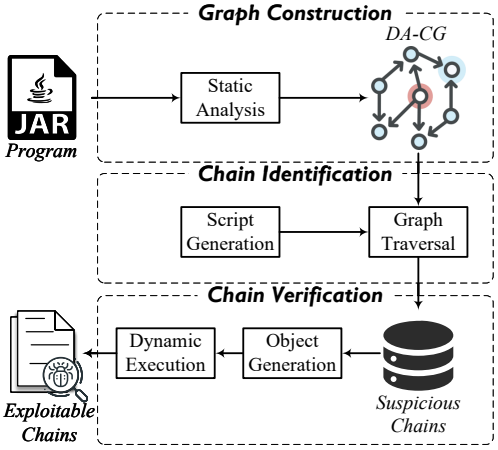


Fig. 4: Framework of *GCMiner*.

Figure 4 shows the framework of *GCMiner*, which contains three modules: *Graph Construction*, *Chain Identification*, and *Chain Verification*. More specifically, *GCMiner* takes a target Java application as the input, and constructs the *Deserialization-Aware Call Graph* (DA-CG) through static analysis to model both explicit and implicit method calls (Section IV-A). Then, *GCMiner* stores the DA-CG into the graph database and searches for suspicious gadget chains through graph traversal (Section IV-B). Finally, to verify whether a statically identified gadget chain is exploitable, *GCMiner* adopts an overriding-guided object generation approach to generate exploitable injection objects for fuzzing (Section IV-C). The gadget chain which receives an injection object reachable to a security-sensitive call site will be confirmed as exploitable.

A. Graph Construction

GCMiner first performs static analysis to generate the *Call Graph* (CG) [23] to capture explicit method calls. Considering that statically constructed CGs may miss some exploitable methods due to their incomplete support for dynamic program features (as discussed in Section III-B), we add additional overriding relations through *Class Hierarchy Analysis* (CHA) [37] to construct a *Deserialization-Aware Call Graph* (DA-CG) to identify implicit method calls. The DA-CG is represented as a directed graph, where methods in each class are graph nodes, and two types of directed relations (i.e., method call and method overriding) between methods are recorded as edges. The introduction of overriding relations contributes to model dynamic behaviors of programs and thereby capturing more exploitable gadgets missed by pure CGs.

It is noteworthy that blindly considering all possible overridden methods on the application's classpath will introduce a large number of false positives. A typical example is `toString()`, one of the common magic methods. For a large Java application, a great number of classes re-implement `toString()` to transform the reference to an object to a user-readable form. To this end, for applications/libraries (e.g., `Apache Commons Collections`) which do not re-implement

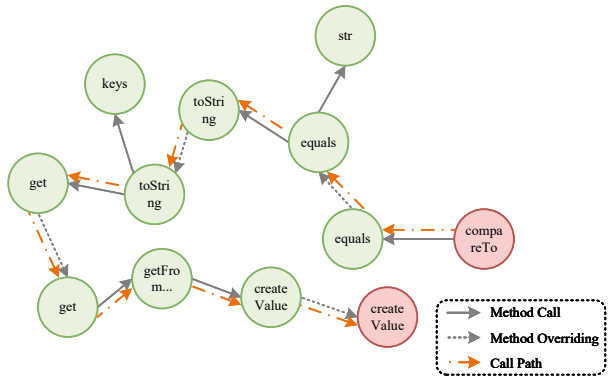


Fig. 5: Partial DA-CG for our motivating example.

their own deserialization operations, we ignore methods whose classes that do not support deserialization operations (i.e., not implementing serialization interfaces like `Serializable`) to focus on deserialization-related method calls because these methods cannot be invoked during object deserialization. For those applications/libraries (e.g., `XStream`) which re-implement their own deserialization operations, we still consider all possible methods.

Example. Figure 5 shows a part of our constructed DA-CG for the motivating example in Figure 1. We can observe that the DA-CG consists of several method nodes (highlighted in green and red) and two types of edges (method call and method overriding). The complete call path (orange shaded) of the gadget chain in the motivating example is clear, which starts from the magic method `CompareTo()` to the security-sensitive call site `Createvalue()`. We can observe that, owing to our additional overriding relations, exploitable gadgets (e.g., `XString.equals()` at line 6 and `MultiUIDefaults.toString()` at line 9 in Figure 1a) can also be identified.

B. Chain Identification

After graph construction, we store the DA-CG into the graph database and search for suspicious gadget chains through customized query scripts.

Specifically, we adopt *Cypher* [38], a declarative language for graph data retrieval [39], to design query scripts for suspicious gadget chain identification. The script mainly includes three components, i.e., start nodes (sources), end nodes (sinks), and path constraints. We use 11 magic methods and 25 security-sensitive call sites found in our empirical study as sources and sinks to limit the retrieval scope. For path constraints, we consider two types of edges (i.e., method call and method overriding) in our DA-CG. Note that, although the overriding relation does not indicate the actual method call, it provides a hint that these overridden methods can be exploited as gadgets to propagate malicious injection objects. Thus, to avoid missing any suspicious gadget chain, we search for all gadget chains between sources and sinks as candidates for verification.

Example. Figure 6 presents an example of our query scripts for suspicious gadget chain identification. We first select

```
1  match (source: Method {NAME:"readObject"})
2  match (sink: Method {NAME:"invoke"})
3  call apoc.algo.allSimplePaths(sink, source, "<Call|Overriding>")
   yield path
4  return path
```

Fig. 6: A simple query script example.

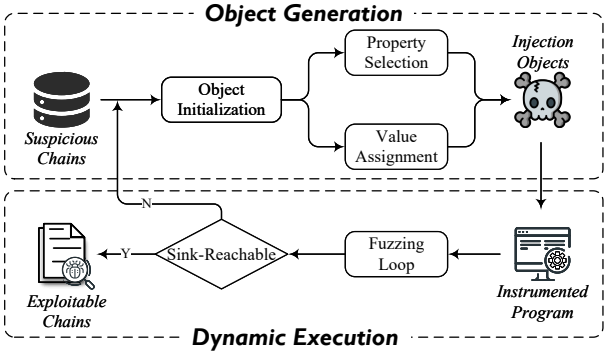


Fig. 7: Overview of gadget chain verification.

the magic methods and security-sensitive call sites we are interested in (line 1-2). For example, we adopt `readObject()` and `invoke()` as the source and sink respectively, and use the built-in method `allSimplePaths()`[4] (line 3) to search for all suspicious gadget chains (i.e., reachable paths from the target source to the sink). A set of paths, which satisfy the following three conditions: 1) its start node is a source method; 2) its end node is a sink method; and 3) the type of connected edges is `CALL` or `Overriding`, is returned as our retrieval results (line 4), i.e., suspicious gadget chains.

C. Chain Verification

Given a set of suspicious gadget chains, *GCMiner* leverages an overriding-guided object generation approach to produce valid injection objects and performs coverage-guided fuzzing for verification. Figure 7 shows the overview of our chain verification, which contains two modules: 1) *Object Generation*; and 2) *Dynamic Execution*. For object generation, *GCMiner* first selects a gadget chain from candidates and instantiates the class to which the given chain's first gadget (i.e., deserialization entry point, or magic method) belongs to construct an initial injection object. Then, to enable the generated injection object to follow the execution flow of the target gadget chain, *GCMiner* modifies the property values of the injection object by dynamic binding. For dynamic execution, *GCMiner* feeds these generated injection objects into the target program for fuzzing. Once an injection object reaches the target security-sensitive call site, this gadget chain under testing will be confirmed as exploitable. The procedure of chain verification will not terminate until all statically identified candidate chains have been checked.

Object Generation. Generating an injection object for a given gadget chain requires dynamically modifying its property values to enable the injection object to follow the execution

[4]https://neo4j.com/developer/neo4j-apoc/

flow of the identified gadget chain to reach security-sensitive call site. To achieve this, we leverage overriding relations between methods to guide injection object generation.

Specifically, *GCMiner* first generates an initial injection object by instantiating the class to which the entry point (i.e., magic method) of the target chain belongs. Then, according to the relation information provided by DA-CG, *GCMiner* modifies the property values of the initial injection object through dynamic binding to ensure that the injection object can reach the dangerous sink. Such a dynamic property value assignment requires 1) selecting a property, which receives a serializable class object as its value, from the property list of the injection object (i.e., *property selection*); and 2) assigning an instantiated object as the value to the corresponding property (i.e., *value assignment*).

For property selection, we dynamically obtain each property type of the injection object by Java reflection [40]–[42], which allows a software system to inspect and change the behavior of its classes, interfaces, methods, and fields at runtime. When a property type is a class object, this property is listed as a candidate for value assignment. These candidate properties may be exploited by attackers to implement insecure deserialization paths. For value assignment, we adopt the overriding edge provided by our constructed DA-CG as guidance to select an overridden method that can be invoked by dynamic binding to enable the injection object to reach the dangerous sink of the given gadget chain. *GCMiner* instantiates the class to which the overridden method belongs, and analyzes whether the class object can be assigned as the property value to a candidate property of the injection object. If the class to which the overridden method belongs is a subclass of the property type of the injection object, *GCMiner* will assign the instance containing the overridden method to the corresponding property of the injection object.

Considering that the properties of a given gadget chain may need multiple modification to facilitate the execution of the gadget chain, we repeat the above process (i.e., property selection and value assignment) for dynamic property value assignment to enable the injection object to reach the security-sensitive call site. Once the property values of the injection object are configured, *GCMiner* will feed it into the target application for fuzzing.

Dynamic Execution. As discussed before, an exploitable gadget chain should enable the malicious deserialized object to reach (in terms of control flow) and affect (in terms of data flow) the security-sensitive sink, while our overriding-guided object generation strategy just assure the reachability of these statically reported gadget chains (i.e., attackers can dynamically invoke these gadgets to inject malicious objects). Hence, to verify whether the dangerous sink of the given gadget chain can be affected by the generated injection object, *GCMiner* adopts a generation-based coverage-guided Java fuzzing framework JQF [43], which provides an extensible interface for users to easily integrate their own input generation mechanism for testing.

Specifically, *GCMiner* first executes the injection object on the instrumented program and collects the code coverage as feedback to guide the fuzzing procedure. The injection object which covers more branches in the gadgets is more likely to reach the target sink. Unlike traditional coverage-guided fuzzing which blindly increases the code coverage to accidentally trigger potential vulnerabilities, *GCMiner* only instruments classes to which gadgets belong on the application's classpath, which makes the fuzzer pay more attention to those seeds (i.e., generated or mutated injection objects) that trigger more code snippets within gadgets. These seeds are more likely to reach dangerous sinks. For mutation, *GCMiner* leverages the JQF-Zest algorithm [44] to produce new inputs that get deeper into the target gadget chain by mutating the interesting seeds at the bit-level. These bit-level mutations correspond to property-level mutations on structured injection objects [45]. For *primitive* data types (e.g., `boolean`, `int`), the fuzzer uses multiple pseudo-random methods built in JQF to convert untyped bit parameters into random typed values. For the *reference* data types, the fuzzer tailors targeted templates for specific types. When the property type is `class`, the fuzzer will randomly select a class from the candidate classes (i.e., sub-classes) of this property via the method `random.choose()`. For an `array` property, the fuzzer uses the method `random.nextInt()` to randomly set up the array size and assigns random values based on the type of elements (i.e., instances that inherit the class type of the array) to the array.

Once a generated injection object reaches the target security-sensitive call site, the suspicious chain under testing will be confirmed as a gadget chain. The procedure of chain verification will not terminate until all the statically reported gadget chains have been checked.

V. Experiment

A. Research Questions

RQ3: Effectiveness of *GCMiner*. How effective is *GCMiner* in mining Java deserialization gadget chains?

By investigating this RQ, we aim to answer how well does *GCMiner* perform in comparison with the state-of-the-art automatic Java deserialization gadget chain mining techniques.

RQ4: Ablation study.

- RQ4a: Impact of additional sources and sinks. Whether these newly added sources and sinks contribute to mining more exploitable gadget chains?

Our approach includes some new magic methods and security-sensitive call sites, which aims to search for more suspicious gadget chains. This question aims to show whether these additional sources and sinks can help find more exploitable gadget chains?

- RQ4b: Impact of introducing method overriding. Does the introduction of overriding relations contribute to identifying more exploitable gadgets?

One of our key insights is to introduce additional overriding relations to capture dynamic program features abused by attackers for gadget chain construction. By investigating this RQ, we aim to show whether the introduction of overriding relations helps identify more exploitable gadget chains.

- RQ4c: Impact of overriding-guided object generation. Can our overriding-guided object generation approach produce valid injection objects for fuzzing?

Another key insight of our approach is that generating exploitable injection objects through binding exploitable gadget dynamically to enable them to reach dangerous sinks. By answering this RQ, we aim to show whether leveraging overriding relations to guide dynamic binding can generate valid injection objects for gadget chain verification.

B. Experiment Setup

1) Benchmark: To evaluate the effectiveness of *GCMiner*, we adopted our manually constructed Java deserialization vulnerability benchmark in Table I, which consists of a widely-used gadget chain collection ysoserial [18] repository and 18 famous Java applications with 86 known gadget chains.

2) Baseline methods: We compared *GCMiner* with a well-known open-source tool, *Gadget Inspector* [10], and a previous study, *Serhybrid* [11].

3) Implementation: We used *Tabby* [46], a Java code analysis tool based on *Soot* [47], to extract both call and overriding relations between methods for DA-CG construction. For chain identification, we used a popular graph database Neo4j [48] to perform our customized query scripts. To verify statically identified gadget chains, we implemented our overriding-guided object generation strategy based on JQF [43], a coverage-guided Java fuzzing framework. JQF was selected for its extensibility in implementing structured seed generation templates. All experiments were conducted on a Linux workstation with an Intel(R) Core(TM) i9-12900k @3.90GHz and 128 GB of RAM, running Ubuntu 18.04.4 LTS with JDK 1.8.0_152.

4) Experimental configurations: For RQ3, we ran *GCMiner* and baselines on vulnerability-specific versions of applications in our benchmark. Unfortunately, despite our best efforts, *Serhybrid* was not reproducible. We made unsuccessful attempts to contact the authors for suggestions. We could only compare *GCMiner* with *Serhybrid* on the results of several applications reported in the original paper. To ensure the fairness of the comparison, we conducted the experiments under the same conditions and evaluated the performance with the same metrics. We repeated each experiment 10 times and reported their average performance [49]. According to the assessment of our employed security experts (each of them had two to five years of vulnerability mining-related experience gained in industry), we empirically set the threshold for the length of each chain to 15 gadgets to avoid the path explosion problem during graph traversal. For each statically identified gadget chain, we limit the fuzzing campaign of *GCMiner* to 120 seconds. For RQ4a, we only used statically constructed CGs for gadget chain identification to evaluate the contribution of overriding relations to capturing exploitable gadgets. In RQ4b, to investigate the contribution of our overriding-guided object generation, we randomly built injection objects based on DA-CG for comparison.

TABLE II: Comparison results between *GCMiner* and *Gadget Inspector*.

Application	#KGC	*GCMiner*			*Gadget Inspector*		
		#TP/#Rep	P*	R	#TP/#Rep	P	R
ysoserial	34	21 / 29	1	0.618	3 / 116	0.026	0.088
JBoss RESTEasy	1	1 / 3	1	1	0 / 2	0	0
Apache Camel	2	2 / 2	1	1	0/ 2	0	0
Apache Brooklyn	1	1 / 1	1	1	0 / 2	0	0
Apache XBean	1	0 / 2	1	0	0 / 2	0	0
Shiro	3	1 / 2	1	0.333	0 / 2	0	0
Pippo	2	2 / 5	1	1	0 / 2	0	0
Adobe Coldfusion	2	2 / 3	1	1	1/ 2	0.500	0.500
VMWare VCenter	1	1 / 1	1	1	0 / 2	0	0
Red5	1	1 / 2	1	1	0/2	0	0
Hessian	5	4 / 7	1	0.800	0 / 2	0	0
XStream	14	12 / 19	1	0.857	1 / 2	0.500	0.071
Commons Collections	3	3 / 7	1	1	0 / 12	0	0
Dubbo	2	1 / 2	1	0.500	0 / 3	0	0
WebLogic	5	4 / 11	1	0.800	0 / 6	0	0
Emissary	3	2 / 4	1	0.667	0 / 3	0	0
Jenkins	2	1 / 9	1	0.500	0 / 2	0	0
Apache OFBiz	3	1 / 4	1	0.333	0 / 2	0	0
Spring	1	1 / 5	1	1	0 / 6	0	0
Total	**86**	**61 / 118**	**1**	**0.709**	**5 / 172**	**0.029**	**0.058**

* Since *GCMiner* adopted fuzzing to verify exploitable gadget chains, we used dynamically confirmed gadget chains as *Rep* to compute the precision.

C. Evaluation Metrics

To evaluate our approach, we used the following metrics.

***Known Gadget Chains* (KGC)** is the number of the publicly known gadget chains in a target application.

***Reported Gadget Chains* (Rep)** computes the total number of gadget chains statically reported by each approach.

***True Positives* (TP)** is the number of truly exploitable gadget chains reported by each approach. In our experimental evaluation, TP counts how many known gadget chains in the benchmark are mined.

***Precision* (P)** is the fraction of truly exploitable gadget chains among the reported ones. It is calculated as: $P = \frac{TP}{Rep}$.

***Recall* (R)** is the fraction of known gadget chains that are identified by each approach. It is calculated as: $R = \frac{TP}{KGC}$.

D. Effectiveness of GCMiner (RQ3)

Table II shows the overall results of *GCMiner*. In total, *GCMiner* identifies 61 out of 86 known gadget chains with a recall of $61/86 = 70.9\%$ without false positives. The reasons for these false negatives mainly come from two aspects. In the static identification stage, due to the limited support for certain dynamic features of Java language such as *reflective calls* [40] and *dynamic proxy* [35], certain exploitable gadgets on the classpath of the application are not captured by our DA-CG. For example, in `Groovy1` [50], the attacker could exploit the class `ConvertedClosure`, whose constructor receives a proxy `MethodClosure` as its parameters, to pass tainted arguments to the gadget `MethodClosure.call()` to execute the malicious commands. Due to the unawareness of which classes can be proxied, gadget chains involving dynamic proxy during their construction are difficult to be identified by *GCMiner*,

```
1  /*com.sun.rowset.JdbcRowSetImpl.class*/
2  public DatabaseMetaData getDatabaseMetaData() throws SQLException {
3    Connection var1 = this.connect();
4    return var1.getgetMetaData();}
5  private Connection connect() throws SQLException { // ...
6    try {
7      InitialContext var1 = new InitialContext();
8      DataSource var2 = (DataSource)var1.lookup(this.getDataSourceName());
9    // ...
10 }
11 /*javax.naming.InitialContext.class*/
12 public Object lookup(String name) throws NamingException {
13   return getURLOrDefaultInitCtx(name).lookup(name);}
```

Fig. 8: A gadget chain identified by *GCMiner* but missed by *Gadget Inspector*.

TABLE III: Comparison results between *GCMiner* and *Serhybrid*.

Application	#KGC	*GCMiner*		*Serhybrid*	
		#Object	#Exploit	#Object	#Exploit
bsh-2.0b5	1	1	0	0	0
clojure-1.8.0	1	2	1	N/A	0
commons-beanutils-1.9.2	1	2	1	0	0
commons-collections-3.1	5	12	3	1	1
commons-collections4-4.0	2	4	2	1	1
groovy-2.3.9	1	2	0	0	0
hibernate	2	3	2	0	0
jython-standalone-2.5.2	1	1	0	N/A	0
rome-1.0	1	2	1	0	0
Total	**15**	**29**	**10**	**2**	**2**

resulting in false negatives. In the dynamic verification stage, due to certain specific constraints, some statically identified gadget chains cannot be dynamically validated by *GCMiner*. For example, *GCMiner* fails to construct sink-reachable injection objects for `AspectJWeaver` [51] in *ysoserial* because its sink method `writeToPath()` receives a file as an input, which is not supported by our injection object generation strategy.

GCMiner vs. *Gadget Inspector*. As shown in Table II, *GCMiner* identifies a total of 118 suspicious gadget chains, of which 61 are known gadget chains. By contrast, *Gadget Inspector* can only identify five exploitable gadget chains with a recall of $5/86 = 5.8\%$, and a precision of $5/172 = 2.9\%$. In addition, all gadget chains identified by *Gadget Inspector* are covered by *GCMiner*. Such a significant performance gap may result from two aspects. On the one hand, constrained by a limited number of exploitable magic methods and security-sensitive call sites, a large number of suspicious gadget chains on the classpath of the application are missed by *Gadget Inspector*. Figure 8 presents a typical example which can be identified by *GCMiner* but missed by *Gadget Inspector*. It is a widely exploited gadget chain which can be triggered to perform JNDIi attack. During the process of object deserialization, the method `getDatabaseMetaData()` (line 2) in class `JdbcRowSetImpl` will invoke the method `connect()` (line 3) by default. To get the context of the object transferred by users, the method `connect()` will invoke the method `lookup()` (line 8), which is a security-sensitive call site that can be exploited by remote attackers to inject malicious code. However, since the method `lookup()` is not considered by *Gadget Inspector*, this gadget chains is missed. On the other hand, due to the limited precision of static analysis, *Gadget Inspector* cannot guarantee that identified gadget chains are truly exploitable, resulting in many *false positives*. Owing to our overriding-guided object generation approach which produces valid injection objects for verification, the gadget chains which cannot be exploited will be filtered out.

GCMiner vs. *Serhybrid*. We also compared *GCMiner* with *Serhybrid* on the selected applications from *ysoserial* repository. Table III presents the comparative results of *GCMiner* and *Serhybrid*. Column "#Object" presents the number of injection objects generated by each approach. Column "#Exploit" is the number of injection objects that can trigger known gadget chains.

The results show that, from the nine applications which contain 15 known gadget chains, *GCMiner* successfully generates 29 injection objects, 10 of which are valid injection objects that can be leveraged by attackers to perform deserialization attack. By contrast, *Serhybrid* generates two valid injection objects. Both two gadget chains reported by *Serhybrid* are covered by *GCMiner*. Although *Serhybrid* achieves higher precision due to its points-to analysis, many gadget chains are missed. The reason may be that the heavy use of Java dynamic program features makes *Serhybrid* hard to compute reachable paths to dangerous sinks. As a result, only a small number of objects can be successfully generated for verification. By contrast, owing to our constructed DA-CG, *GCMiner* can better model complex dynamic behaviors of programs and contribute to capturing more exploitable gadgets. Besides, from the results, we can observe that *Serhybrid* is hard to generate valid injection objects (timeout even occurred when analyzing `Clojure` [52] and `Jython` [53]) for verification. Such situation occurs due to the strict object generation strategy of *Serhybrid* that relies on the analysis results of heap access paths. Therefore, once the precise execution paths which perform malicious objects to the target sink cannot be identified by static analysis, *Serhybrid* is difficult to produce valid injection objects.

Answer to RQ3: *GCMiner significantly outperforms the state-of-the-art Java deserialization gadget chain mining tools, identifying 56 unique gadget chains that cannot be identified by baselines.*

E. Ablation study (RQ4)

1) ***RQ4a: Impact of additional sources and sinks****:* Table IV summarizes the number of known gadget chains discovered by *GCMiner* and other variants. Overall, with these additional sources and sinks newly collected in our previous empirical study, *GCMiner* identifies 27 more probably exploitable gadget chains, 12 of which are the known gadget chains. Besides, we also notice that the improvement of *Gadget Inspector*$_{Var}$ is not significant. Despite of additional sources and sinks, *Gadget Inspector*$_{Var}$ only mines two more exploitable gadget chains but reports 808 more false positives. The reason may be that the introduction of new exploitable sources and gadgets can improve the search scope about suspicious gadget chains to a certain extent. Meanwhile, it also amplifies the deficiencies

TABLE IV: Impact of additional sources and sinks on gadget chain mining.

Application	#KGC	*GCMiner*		$GCMiner_{Var}$		$Gadget\ Inspector_{Var}$	
		#Rep	#TP	#Rep	#TP	#Rep	#TP
ysoserial	34	29	21	24	15	637	4
JBoss RESTEasy	1	3	1	2	1	14	0
Apache Camel	2	2	2	2	2	14	0
Apache Brooklyn	1	1	1	1	1	16	0
Apache XBean	1	2	0	1	0	14	0
Shiro	3	2	1	1	0	14	0
Pippo	2	5	2	3	1	14	0
Adobe Coldfusion	2	3	2	3	2	14	1
VMWare VCenter	1	1	1	1	1	12	0
Red5	1	2	1	1	1	14	0
Hessian	5	7	4	5	3	14	0
XStream	14	19	12	15	10	14	2
Commons Collections	3	7	3	7	3	69	0
Dubbo	2	2	1	2	1	16	0
WebLogic	5	11	4	8	3	21	0
Emissary	3	4	2	3	2	11	0
Jenkins	2	9	1	6	1	14	0
Apache OFBiz	3	4	1	2	1	14	0
Spring	1	5	1	4	1	46	0
Total	**86**	**118**	**61**	**91**	**49**	**982**	**7**

in imprecision of static analysis in gadget chain mining. By contrast, our reflection-guided exploit generation approach can effectively filter out invalid gadget chains.

Answer to RQ4a: *Additional exploitable magic methods and security-sensitive call sites are useful to identify more potential gadget chains.*

*2) **RQ4b: Impact of introducing method overriding**:* Table V shows the number of gadget chains identified by *GCMiner* in different configurations. Columns "With Overriding" and "W/O Overriding" represent *GCMiner* enabling/disabling method overriding in DA-CG, respectively. The results demonstrate that the introduction of method overriding positively contributes to improving the effectiveness of *GCMiner*. In particular, by taking method overriding into consideration, *GCMiner* can identify 118 suspicious gadget chains, of which 61 are true positives. By contrast, *GCMiner* based on statically constructed CGs can only identify 3 out 9 exploitable gadget chains. Hence, overriding relations can effectively capture implicit method invocations and is helpful for mining exploitable gadget chains.

Answer to RQ4b: *The introduction of overriding relations significantly enhances the capability of existing static analysis in capturing potential exploitable gadgets, enabling our approach to identify more exploitable gadget chains.*

*3) **RQ4c: Impact of overriding-guided object generation**:* Table VI presents the results of *GCMiner* and its corresponding variant ($GCMiner_{NG}$) which generates injection objects with no guidance during fuzzing. The results show that our overriding-guided object generation can effectively generate injection objects, 61 of which are valid exploits. By contrast, $GCMiner_{NG}$ can only generate seven objects, none of which can reach the dangerous sink. This performance gap may be due to that constrained by the highly structured characteristic

TABLE V: Impact of introducing overriding relations on gadget chain identification.

Application	#KGC	*With Overriding*		*W/O Overriding*	
		#Rep	#TP	#Rep	#TP
ysoserial	34	29	21	6	2
JBoss RESTEasy	1	3	1	0	0
Apache Camel	2	2	2	1	0
Apache Brooklyn	1	1	1	0	0
Apache XBean	1	2	0	0	0
Shiro	3	2	1	0	0
Pippo	2	5	2	1	0
Adobe Coldfusion	2	3	2	0	0
VMWare VCenter	1	1	1	0	0
Red5	1	2	1	0	0
Hessian	5	7	4	0	0
XStream	14	19	12	3	0
Commons Collections	3	7	3	2	1
Dubbo	2	2	1	0	0
WebLogic	5	11	4	1	0
Emissary	3	4	2	0	0
Jenkins	2	9	1	1	0
Apache OFBiz	3	4	1	0	0
Spring	1	5	1	0	0
Total	**86**	**118**	**61**	**9**	**3**

TABLE VI: Impact of overriding-guided object generation on gadget chain verification.

Application	#KGC	*GCMiner*		$GCMiner_{NG}$	
		#Object	#Exploit	#Object	#Exploit
ysoserial	34	86	21	5	0
JBoss RESTEasy	1	3	1	0	0
Apache Camel	2	7	2	0	0
Apache Brooklyn	1	3	1	0	0
Apache XBean	1	2	0	0	0
Shiro	3	6	1	0	0
Pippo	2	5	2	0	0
Adobe Coldfusion	2	7	2	0	0
VMWare VCenter	1	3	1	0	0
Red5	1	2	1	0	0
Hessian	5	11	4	0	0
XStream	14	48	12	1	0
Commons Collections	3	8	3	1	0
Dubbo	2	4	1	0	0
WebLogic	5	13	4	0	0
Emissary	3	9	2	0	0
Jenkins	2	3	1	0	0
Apache OFBiz	3	5	1	0	0
Spring	1	4	1	0	0
Total	**86**	**229**	**61**	**7**	**0**

(i.e., the property values of an exploitable object need to be modified many times to reach the dangerous sink) of the injection object, existing object generation techniques can hardly produce valid injection objects for verification.

Answer to RQ4c: *With our overriding-guided object generation, GCMiner can effectively generate sink-reachable injection objects for fuzzing.*

VI. Threats to Validity

A. External validity

A main external threat comes from the generalization of our empirical results. Similar to existing works [6], [10], [11], [54], the effectiveness (recall) of our static gadget chain

identification also relies heavily on the prior expert knowledge of available sources and sinks. Considering that there are a few orthogonal tools/approaches [55], [56] have been proposed to automatically identify untrusted deserialization entry points, and our knowledge base is configurable, i.e., newly disclosed sources and sinks can be dynamically added, the capability to detect unknown Java deserialization vulnerabilities in the wild can be improved. Furthermore, since we only focus on Java deserialization vulnerabilities, all findings and evaluation results may not be applicable to other programming languages (e.g., PHP [54] and .NET [32]) which also suffer from the risk of deserialization vulnerabilities. We leave it as a future work to extend our approach to other languages.

B. Internal validity

Internal validity in our experiment comes from two aspects. On the one hand, since our approach aims to identify exploitable gadget chains instead of detecting Java deserialization vulnerabilities, gadget chains which cannot be exploited in practice may be wrongly reported. To avoid the bias in our conclusions, we tried our best to manually reproduce each vulnerability in our benchmark to make sure the practical exploitability of gadget chains. On the other hand, due to the non-reproducibility of *Serhybrid*, we directly compared our approach with *Serhybrid* on its reported applications and results under the same experimental situations.

VII. Related Work

Java Deserialization Vulnerability Detection. Many studies have been proposed for analyzing, defensing, and detecting Java deserialization vulnerabilities [57]–[61]. Muñoz et al. [33] conducted a comprehensive analysis on JSON deserialization libraries and presented several mitigation measures as takeaways. Carettoni [62] presented a configurable Java deserialization library, which supports multiple optional settings such as blacklist and whitelist, to secure application from untrusted input. Koutroumpouchos et al. [56] proposed an extendable tool *ObjectMap*, which generates a series of requests to validate whether the payload can be directly passed to the target application. Cristalli et al. [63] proposed a dynamic approach, which collects the behavior information of benign deserialization process and constructs the precise execution path to prevent untrusted data input. Sayar et al. [22] conducted a large-scale empirical study on publicly known Java deserialization RCE exploits and investigated how deserialization vulnerabilities manifest in real code bases and libraries.

In order to automatically mine suspicious gadget chains, Haken [10] presented *Gadget Inspector*, which leverages static taint analysis and simple symbolic execution to mine the propagation paths of parameters within/between methods of a target application, and then performs a Breadth-first search (BFS) to search for exploitable gadget chains. Rasheed et al. [11] proposed *Serhybrid*, a hybrid analysis-based approach which constructs a heap abstraction to produce actual input objects to automatically verify exploitable gadget chains. In this paper, *GCMiner* constructs DA-CG to identify more exploitable gadget chains, and leverages an overriding-guided object generation approach to produce valid injection objects.

Automatic Exploit Generation. Automatic Exploit Generation (AEG) is proposed to automatically construct exploits to evaluate the exploitability of vulnerabilities [27], [28], [64], [65]. Avgerinos et al. [26] proposed an automatic vulnerability mining and exploitation approach, which uses program verification to find the input that can be used and make the program enter an unsafe state (such as Out-of-bounds write and malicious format string). Padaryan et al. [66] presented a framework, which does not require debug information and could be applied to binary programs, based on program dynamic analysis and symbol execution to construct exploits for stack buffer overflow vulnerabilities. Wu et al. [67] proposed an automated exploitation framework for kernel *Use-After-Free* (UAF) vulnerabilities. They leveraged fuzzing to provide more kernel crashes on contextual environments as a basis for vulnerability exploitation, and then used symbolic execution to exploit the target vulnerability in different contextual environments. In this paper, based on our constructed DA-CG, *GCMiner* dynamically binds exploitable overridden methods to generate valid injection objects for automatic verification.

VIII. Conclusion

Java deserialization vulnerability receives little attention in the academic community despite its severe impact in practice. In this paper, we call for attention to this problem and performs an empirical study to investigate the characteristics of Java deserialization vulnerabilities from the perspective of gadget chain exploitation. Based on our empirical findings, we propose *GCMiner*, a novel gadget chain mining approach which analyzes both explicit and implicit methods calls to identify more exploitable gadget chains and generates valid injection objects through dynamic binding for fuzzing. The evaluation results show that *GCMiner* significantly outperforms state-of-the-art solutions, and discovers 56 unique gadget chains that cannot be identified by baselines.

In the future, we plan to apply *GCMiner* to the industrial scenario to perform a large-scale case study for evaluation. In addition, we also plan to investigate automatic exploit generation techniques for vulnerability reproduction and confirmation.

Acknowledgment

This work is supported by the National Natural Science Foundation of China (No.61972335, No. 62202414, No.62002309, No. 62272400); the CCF-AFSG Research Fund (No.CCF-AFSG RF20210022); the Six Talent Peaks Project in Jiangsu Province (No. RJFW-053), the Jiangsu "333" Project; the Open Funds of State Key Laboratory for Novel Software Technology of Nanjing University (No.KFKT2022B17), Yangzhou University Top-level Talents Support Program (2019); Xiamen Youth Innovation Fund (3502Z20206036).

References

[1] M. Herlihy and B. Liskov, "A value transmission method for abstract data types," *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 4, pp. 527–551, 1982.

[2] J. C. S. Santos, R. A. Jones, C. Ashiogwu, and M. Mirakhorli, "Serialization-aware call graph construction," in *Proceedings of the 10th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis (SOAP)*. ACM, 2021, pp. 37–42.

[3] Y. Wei, X. Sun, L. Bo, S. Cao, X. Xia, and B. Li, "A comprehensive study on security bug characteristics," *J. Softw. Evol. Process.*, vol. 33, no. 10, 2021.

[4] X. Sun, X. Peng, K. Zhang, Y. Liu, and Y. Cai, "How security bugs are fixed and what can be improved: an empirical study with mozilla," *Sci. China Inf. Sci.*, vol. 62, no. 1, pp. 19 102:1–19 102:3, 2019.

[5] Svoboda, "Exploiting java deserialization for fun and profit," 2016.

[6] J. Dahse, N. Krein, and T. Holz, "Code reuse attacks in PHP: automated POP chain generation," in *Proceedings of the 21th ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2014, pp. 42–53.

[7] Spring4Shell, 2022, https://www.picussecurity.com/resource/spring4shell-spring-core-remote-code-execution-vulnerability.

[8] Spring, 2022, https://spring.io.

[9] OWASP Top Ten 2017 A8: Insecure Deserialization, 2022, https://owasp.org/www-project-top-ten/2017/A8_2017-Insecure_Deserialization.

[10] I. Haken, "Automated discovery of deserialization gadget chains," in *Proceedings of the Black Hat USA*, 2018.

[11] 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 (ASE)*. IEEE, 2020, pp. 1209–1213.

[12] L. Luo, F. Pauck, G. Piskachev, M. Benz, I. Pashchenko, M. Mory, E. Bodden, B. Hermann, and F. Massacci, "Taintbench: Automatic real-world malware benchmarking of android taint analyses," *Empir. Softw. Eng.*, vol. 27, no. 1, p. 16, 2022.

[13] M. Benz, E. K. Kristensen, L. Luo, N. P. Borges, E. Bodden, and A. Zeller, "Heaps'n leaks: how heap snapshots improve android taint analysis," in *Proceedings of the 42nd International Conference on Software Engineering (ICSE)*. ACM, 2020, pp. 1061–1072.

[14] V. J. M. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The art, science, and engineering of fuzzing: A survey," *IEEE Trans. Software Eng.*, vol. 47, no. 11, pp. 2312–2331, 2021.

[15] M. Böhme, C. Cadar, and A. Roychoudhury, "Fuzzing: Challenges and reflections," *IEEE Softw.*, vol. 38, no. 3, pp. 79–86, 2021.

[16] E. Ruf, "Context-insensitive alias analysis reconsidered," in *Proceedings of the 16th ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*. ACM, 1995, pp. 13–22.

[17] M. Reif, F. Kübler, M. Eichberg, D. Helm, and M. Mezini, "Judge: identifying, understanding, and evaluating sources of unsoundness in call graphs," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2019, pp. 251–261.

[18] YSoSerial, 2022, https://github.com/frohoff/ysoserial.

[19] R. E. Gantenbein and D. W. Jones, "The design and implementation of a dynamic binding feature for a high-level language," *J. Syst. Softw.*, vol. 8, no. 4, pp. 259–273, 1988.

[20] S. Esser, "Utilizing code reuse/rop in php application exploits," in *Proceedings of the Black Hat USA*, 2010.

[21] XStream, 2022, https://x-stream.github.io/security.html.

[22] I. Sayar, A. Bartel, E. Bodden, and Y. Le Traon, "An in-depth study of java deserialization remote-code execution exploits and vulnerabilities," *ACM Trans. Softw. Eng. Methodol.*, 2022.

[23] B. G. Ryder, "Constructing the call graph of a program," *IEEE Trans. Software Eng.*, vol. 5, no. 3, pp. 216–226, 1979.

[24] L. Sui, J. Dietrich, A. Tahir, and G. Fourtounis, "On the recall of static call graph construction in practice," in *Proceedings of the 42nd International Conference on Software Engineering (ICSE)*. ACM, 2020, pp. 1049–1060.

[25] M. Nachtigall, M. Schlichtig, and E. Bodden, "A large-scale study of usability criteria addressed by static analysis tools," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2022, pp. 532–543.

[26] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley, "AEG: automatic exploit generation," in *Proceedings of the 18th Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2011.

[27] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing mayhem on binary code," in *Proceedings of the 33rd IEEE Symposium on Security and Privacy (SP)*. IEEE, 2012, pp. 380–394.

[28] H. Hu, Z. L. Chua, S. Adrian, P. Saxena, and Z. Liang, "Automatic generation of data-oriented exploits," in *Proceedings of the 24th USENIX Security Symposium, (USENIX Security)*. USENIX Association, 2015, pp. 177–192.

[29] NVD, 2022, https://nvd.nist.gov.

[30] CVE, 2022, https://cveform.mitre.org.

[31] Exploit-DB, 2022, https://www.exploit-db.com.

[32] M. Shcherbakov and M. Balliu, "Serialdetector: Principled and practical exploration of object injection vulnerabilities for the web," in *Proceedings of the 28th Annual Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2021.

[33] A. Muñoz and O. Mirosh, "Friday the 13th: Json attacks," in *Proceedings of the Black Hat USA*, 2017.

[34] M. Sharp and A. Rountev, "Static analysis of object references in rmi-based java software," *IEEE Trans. Software Eng.*, vol. 32, no. 9, pp. 664–681, 2006.

[35] G. Fourtounis, G. Kastrinis, and Y. Smaragdakis, "Static analysis of java dynamic proxies," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2018, pp. 209–220.

[36] CommonsCollections1, 2022, https://github.com/frohoff/ysoserial/blob/master/src/main/java/ysoserial/payloads/CommonsCollections1.java.

[37] J. Dean, D. Grove, and C. Chambers, "Optimization of object-oriented programs using static class hierarchy analysis," in *Proceedings of the 9th European Conference on Object-Oriented Programming (ECOOP)*. Springer, 1995, pp. 77–101.

[38] R. Angles, "A comparison of current graph database models," in *Proceedings of the 28th International Conference on Data Engineering (ICDE)*. IEEE, 2012, pp. 171–177.

[39] X. Cheng, X. Sun, L. Bo, and Y. Wei, "KVS: a tool for knowledge-driven vulnerability searching," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 2022, pp. 1731–1735.

[40] Y. Li, T. Tan, and J. Xue, "Understanding and analyzing java reflection," *ACM Trans. Softw. Eng. Methodol.*, vol. 28, no. 2, pp. 7:1–7:50, 2019.

[41] B. Foote and R. E. Johnson, "Reflective facilities in smalltalk-80," in *Proceedings of the 4th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA)*. ACM, 1989, pp. 327–335.

[42] B. C. Smith, "Reflection and semantics in lisp," in *Proceedings of the 11th Annual ACM Symposium on Principles of Programming Languages (POPL)*. ACM, 1984, pp. 23–35.

[43] R. Padhye, C. Lemieux, and K. Sen, "JQF: coverage-guided property-based testing in java," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2019, pp. 398–401.

[44] R. Padhye, C. Lemieux, K. Sen, M. Papadakis, and Y. L. Traon, "Semantic fuzzing with zest," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2019, pp. 329–340.

[45] Z. Zhou, L. Bo, X. Wu, X. Sun, T. Zhang, B. Li, J. Zhang, and S. Cao, "SPVF: security property assisted vulnerability fixing via attention-based models," *Empir. Softw. Eng.*, vol. 27, no. 7, p. 171, 2022.

[46] X. Chen, B. Wang, Z. Jin, Y. Feng, X. Li, X. Feng, and Q. Liu, "Tabby: Automated gadget chain detection for java deserialization vulnerabilities," in *Proceedings of the 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Network (DSN)*. IEEE, 2023.

[47] Soot, 2022, https://soot-oss.github.io/soot.

[48] Neo4j, 2022, https://neo4j.com.

[49] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the 25th ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2018, pp. 2123–2138.

[50] Groovy1, 2022, https://github.com/frohoff/ysoserial/blob/master/src/main/java/ysoserial/payloads/Groovy1.java.

[51] AspectJWeaver, 2022, https://github.com/frohoff/ysoserial/blob/master/src/main/java/ysoserial/payloads/AspectJWeaver.java.

[52] Clojure, 2022, https://github.com/frohoff/ysoserial/blob/master/src/main/java/ysoserial/payloads/Clojure.java.
[53] Jython, 2022, https://github.com/frohoff/ysoserial/blob/master/src/main/java/ysoserial/payloads/Jython1.java.
[54] S. Park, D. Kim, S. Jana, and S. Son, "FUGIO: Automatic exploit generation for PHP object injection vulnerabilities," in *Proceedings of the 31th USENIX Security Symposium (USENIX Security)*. USENIX Association, 2022.
[55] Java Deserialization Scanner, 2022, https://github.com/federicodotta/Java-Deserialization-Scanner.
[56] 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, (PCI)*. ACM, 2019, pp. 67–72.
[57] B. A. Azad, P. Laperdrix, and N. Nikiforakis, "Less is more: Quantifying the security benefits of debloating web applications," in *Proceedings of the 28th USENIX Security Symposium (USENIX Security)*. USENIX Association, 2019.
[58] S. Cao, X. Sun, L. Bo, R. Wu, B. Li, and C. Tao, "MVD: memory-related vulnerability detection based on flow-sensitive graph neural networks," in *Proceedings of the 44th IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. ACM, 2022, pp. 1456–1468.
[59] S. Cao, X. Sun, L. Bo, Y. Wei, and B. Li, "*BGNN4VD*: Constructing bidirectional graph neural-network for vulnerability detection," *Inf. Softw. Technol.*, vol. 136, p. 106576, 2021.
[60] T. Zhou, X. Sun, X. Xia, B. Li, and X. Chen, "Improving defect prediction with deep forest," *Inf. Softw. Technol.*, vol. 114, pp. 204–216, 2019.
[61] F. Subhan, X. Wu, L. Bo, X. Sun, and M. Rahman, "A deep learning-based approach for software vulnerability detection using code metrics," *IET Softw.*, vol. 16, no. 5, pp. 516–526, 2022.
[62] Carettoni, "Defending against java deserialization vulnerabilities," 2016.
[63] S. Cristalli, E. Vignati, D. Bruschi, and A. Lanzi, "Trusted execution path for protecting java applications against deserialization of untrusted data," in *Proceedings of the 21st International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*. Springer, 2018, pp. 445–464.
[64] A. Kiezun, P. J. Guo, K. Jayaraman, and M. D. Ernst, "Automatic creation of SQL injection and cross-site scripting attacks," in *Proceedings of the 31st International Conference on Software Engineering (ICSE)*. IEEE, 2009, pp. 199–209.
[65] A. Alhuzali, R. Gjomemo, B. Eshete, and V. N. Venkatakrishnan, "NAVEX: precise and scalable exploit generation for dynamic web applications," in *Proceedings of the 27th USENIX Security Symposium (USENIX Security)*. USENIX Association, 2018, pp. 377–392.
[66] V. A. Padaryan, V. Kaushan, and A. Fedotov, "Automated exploit generation for stack buffer overflow vulnerabilities," *Programming and Computer Software*, vol. 41, no. 6, pp. 373–380, 2015.
[67] W. Wu, Y. Chen, J. Xu, X. Xing, X. Gong, and W. Zou, "FUZE: towards facilitating exploit generation for kernel use-after-free vulnerabilities," in *Proceedings of the 27th USENIX Security Symposium (USENIX Security)*. USENIX Association, 2018, pp. 781–797.