ODDFuzz: Discovering Java Deserialization Vulnerabilities via Structure-Aware Directed Greybox Fuzzing

Sicong Cao[†*], Biao He[‡], Xiaobing Sun[†✉], Yu Ouyang[‡], Chao Zhang[§], Xiaoxue Wu[†], Ting Su[¶],
Lili Bo[†], Bin Li[†], Chuanlei Ma[‡], Jiajia Li[‡], Tao Wei[‡]

[†]Yangzhou University   [‡]Ant Group   [§]Tsinghua University   [¶]East China Normal University

[†]{DX120210088, xbsun, xiaoxuewu, lilibo, lb}@yzu.edu.cn,
[‡]{hb187361, yu.oyy, chuanlei.mchl, jiajia.lijj, lenx.wei}@antgroup.com,
[§]chaoz@tsinghua.edu.cn, [¶]tsu@sei.ecnu.edu.cn

Abstract—**Java deserialization vulnerability is a severe threat in practice. Researchers have proposed static analysis solutions to locate candidate vulnerabilities and fuzzing solutions to generate proof-of-concept (PoC) serialized objects to trigger them. However, existing solutions have limited effectiveness and efficiency.**

In this paper, we propose a novel hybrid solution ODDFuzz to efficiently discover Java deserialization vulnerabilities. First, ODDFuzz performs lightweight static taint analysis to identify candidate *gadget chains* that may cause deserialization vulnerabilities. In this step, ODDFuzz tries to locate all candidates and avoid false negatives. Then, ODDFuzz performs directed greybox fuzzing (DGF) to explore those candidates and generate PoC testcases to mitigate false positives. Specifically, ODDFuzz applies a structure-aware seed generation method to guarantee the validity of the testcases, and adopts a novel hybrid feedback and a step-forward strategy to guide the directed fuzzing.

We implemented a prototype of ODDFuzz and evaluated it on the popular Java deserialization repository *ysoserial*. Results show that, ODDFuzz could discover 16 out of 34 known gadget chains, while two state-of-the-art baselines only identify three of them. In addition, we evaluated ODDFuzz on real-world applications including `Oracle WebLogic Server`, `Apache Dubbo`, `Sonatype Nexus`, and `protostuff`, and found six previously unreported exploitable gadget chains with five CVEs assigned.

I. INTRODUCTION

The serialization mechanism [1], which is supported by mainstream programming languages like Java, JavaScript, PHP, and .NET, enables an application to convert an object to a stream of bytes for cross-process or cross-platform data transmission and persistence storage [2]. The counterpart of serialization is deserialization, which reconstructs an object from a serialized byte stream. This deserialization process is *dynamic*, as different objects lead to polymorphic runtime behaviors. Advanced language features (e.g., Java reflection [3]) make the process even more dynamic. This process is also *open*, i.e., crafted serialized objects may be injected by adversaries, which breaks the traditional trust boundary of inter-process data transmission and introduces attack surfaces.

Applications that unsafely deserialize incoming serialized objects would be abused to invoke a series of methods on the classpath, named *gadget chains*, and eventually hijack security-sensitive code (e.g., `Method.invoke()`) or cause other consequences (e.g., access control bypass) [4], [5]. Such *open dynamic deserialization* (ODD) vulnerabilities are prevalent and devastating. The past few years have seen a proliferation of deserialization attacks in famous Java applications. For example, a recent zero-day vulnerability (named `Spring4Shell` [6]) discovered in the `Spring` Framework [7] allows an attacker to send a specially crafted HTTP request to bypass protections in the library's HTTP request parser, leading to remote code execution (RCE). Due to the dominance of `Spring` framework in the Java ecosystem, a large number of applications could potentially be impacted.

A limited number of tools [8]–[13] have been proposed to discover ODD vulnerabilities in Java applications. The root cause of ODD vulnerabilities is that, the deserialized objects can *reach* (in terms of control flow) and *affect* (in terms of data flow) the sensitive code (sinks) of target applications. Therefore, a straightforward way to discover ODD vulnerabilities is static taint analysis, as GadgetInspector [12] does. However, such a purely static solution may suffer precision issues due to the limited support for Java deserialization-related features [2], [14], resulting in both high false-negative and high false-positive rates. Furthermore, it requires manual inspection of the reports, which is *time-consuming* and *error-prone*. To alleviate this problem, SerHybrid [13] adopts a hybrid analysis solution, which analyzes the heap access paths to find source objects that affect security-sensitive call sites, and utilizes fuzzing [15], [16] to generate source injection objects to verify whether the sinks are reachable.

However, these solutions in general have limited effectiveness and efficiency due to three challenges. First, existing static analysis solutions struggle to make trade-offs between precision and recall. Due to the runtime polymorphism of Java language, any available overridden method (gadget) on the application's classpath may be exploited to construct gadget chains. Given that blindly enumerating all possible gadget chains will inevitably suffer from the path explosion problem, existing solutions often employ taint analysis [17] to prune infeasible gadget chains. However, they either are prone to precision issues [12], or may not work due to the huge computation space caused by the prohibitive number of candidate gadget chains [18]. Second, existing fuzzing solutions are *ineffective* at generating testcases (i.e., injection objects) to reach sinks. Note that, the injection objects may have a multilevel class hierarchy and their properties should satisfy certain control-flow or data-flow constraints. Fuzzing solutions

without prior knowledge about such a complex nested form of structures are ineffective at generating qualified objects. Third, existing fuzzing solutions are *inefficient* at generating testcases to reach sinks. They are *coverage-guided* (i.e., trying to cover more code) rather than *target-directed* (i.e., trying to reach specific code sooner), thus wasting too much energy on program paths that will not reach sinks.

In summary, Java is one of the most popular language suffering devastating ODD vulnerabilities [19], but there are few solutions to discover Java ODD vulnerabilities while existing solutions have limited efficiency and effectiveness. To address these challenges, we propose a novel hybrid solution ODDFUZZ to discover ODD vulnerabilities for Java applications. In particular, ODDFUZZ performs a lightweight taint analysis, which makes a trade-off between precision and recall when handling Java runtime polymorphism, to identify possible candidate gadgets chains. Then, ODDFUZZ models the data constraints of such gadget chains as a tree and utilizes it to perform structure-aware fuzzing. Finally, ODDFUZZ adopts a novel directed fuzzing solution driven by a step-forward mutation strategy and a hybrid feedback, to reach candidate vulnerabilities rapidly.

We implemented ODDFUZZ based on a popular Java fuzzing framework JQF [20] and evaluated it on ysoserial [8] which is a famous Java deserialization repository with 34 known gadget chains. As the evaluation results show, ODD-FUZZ can identify 16 exploitable gadget chains without false positives, while two state-of-the-art solutions GadgetInspector and SerHybrid only respectively found three and two of them. ODDFUZZ also identifies six previously unknown exploitable gadget chains in four popular Java applications. We have reported these vulnerabilities to the vendors and are working with them on fixing these vulnerabilities. In total, five of these six vulnerabilities have been assigned with new CVEs.

In summary, this paper makes the following contributions:

- We propose a novel solution to Java ODD vulnerability discovery, i.e., ODDFUZZ, which adopts a lightweight taint analysis to identify as many gadget chains as possible and a directed fuzzing solution to validate true positive chains.
- We propose a step-forward and a structure-aware scheme to efficiently guide directed fuzzing towards sensitive sinks.
- We discovered and responsibly report six previously unknown exploitable gadget chains (i.e., ODD vulnerabilities).
- We will open source our tool ODDFUZZ[1] to facilitate further research.

II. BACKGROUND

A. Open Dynamic Deserialization

Open Dynamic Deserialization (ODD), also known as *Object Injection Vulnerabilities* (OIVs) or insecure deserialization [21], refers to a security-critical bug that allows an attacker to manipulate serialized objects to inject harmful data into the application code. This insecure deserialization behavior enables diverse attacks, including denial of service

---

```
1  public class PriorityQueue<E> implements Serializable {
2    transient Object[] queue;
3    private final Comparator<? super E> comparator;
4    private void readObject(java.io.ObjectInputStream s)
                           {...} /*Source*/
5    private void heapify() {...} /*2nd Gadget*/
6    private void siftDown(int k, E x) { /*3rd Gadget*/
7      if (comparator != null)
8        siftDownUsingComparator(k, x);
9    ...}
10   private void siftDownUsingComparator(int k, E x) { /*4th Gadget/
11     if (right < size &&
12       comparator.compare((E) c, (E) queue[right]) > 0)
13   ...}}
14 /*org.apache.commons.collections4.comparators*/
15 public class TransformingComparator<I, O>
             implements Comparator<I>, Serializable {
16   private final Transformer<? super I, ? extends O> transformer;
17   public int compare(I obj1, I obj2) {   /*5th Gadget*/
18     Object value2 = this.transformer.transform(obj2);
19   ...}}
20 /*org.apache.commons.collections4.functors*/
21 public class InvokerTransformer<I, O>
             implements Transformer<I, O>, Serializable {
22   public Object transform(Object input) { /*6th Gadget/
23     Class<?> cls = input.getClass();
24     Method method = cls.getMethod(this.iMethodName,
                           this.iParamTypes);
25     return method.invoke(input, this.iArgs); /*Sink*/
26 }}
```
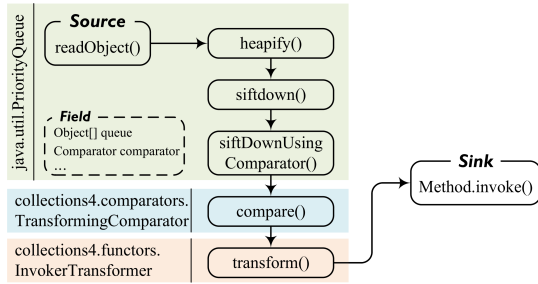
Fig. 1: An exemplary Java ODD vulnerability.

(DoS) attacks, or even remote code execution (RCE) [22]. ODD occurs not only in Java, but also in other mainstream programming languages like JavaScript [23], PHP [18], [24], and .NET [25].
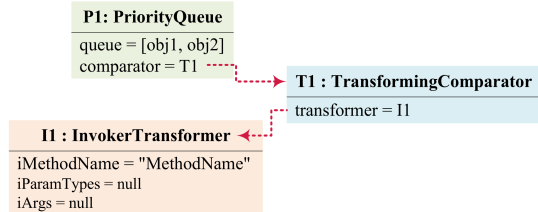
Object Deserialization. Object serialization is a dynamic process of converting objects into a flatter format that can be sent and received as a sequential stream of bytes, for cross-platform data transmission and persistence storage. Object deserialization is the exact opposite of serialization, that is, restoring this byte stream to the original object. In other words, the object's properties are preserved along with their assigned values in the process of serialization and deserialization.

Such a deserialization mechanism is *open*, i.e., allowing arbitrary objects to be deserialized, and *dynamic*, i.e., able to invoke polymorphic methods or reflection-based behaviors and explore diversified paths. These two features can introduce serious ODD vulnerabilities [26]. Typically, deserialized objects are assumed to be trustworthy after some checks. However, a large Java application may implement different libraries with their own dependencies. For developers, this creates a massive pool of classes and methods that are difficult to manage securely, because it is hard to predict which methods can be invoked by the malicious data due to the dynamic nature of Java. From the attacker's perspective, deserializing data from any provenance provides an entry point to an object injection attack, if an attacker is able to chain code fragments of the application together (and execute them in order) and passes data to a security-sensitive call site. Such a code fragment chain is called a *gadget chain*, and each code fragment of this chain is called a *gadget*. Figure 1 shows a simplified code snippet of `CommonsCollections2`, a well-known gadget chain in `Apache Commons Collections4` (ACC) library [8], which enables remote code execution.

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

(b) An injection object triggering `Method.invoke()`.

Fig. 2: Constructing injection objects with POP.

Fig. 3: Threat model.

Fig. 4: Workflow of directed greybox fuzzing.

Property-Oriented Programming. To exploit ODD vulnerabilities, adversaries have to carefully set the properties of the injection object, to chain multilevel objects of specific classes and set certain fields to specific data values, in order to invoke specific polymorphic methods and pass data to security-sensitive call sites. Such a technique used in constructing this injection object is called *Property-Oriented Programming* (POP) [27]. POP allows an attacker to manipulate the data and control flow of the application, thereby exploiting attacker-controllable gadgets on the application's classpath for deserialization attacks.

Figure 2 depicts an example of how an attacker constructs a malicious injection object with POP to exploit the ODD vulnerability shown in Figure 1, where Figure 2a presents its corresponding stack trace. An attacker instantiates an injection object `PriorityQueue`, which contains a malicious payload within its field `queue` (line 2), for exploitation. At the bottom of Figure 2a are two exploitable gadgets, `compare()` (line 17) and `transform()` (line 22) in ACC, required to trigger the security-sensitive call site `Method.invoke()` (line 25). To enable the injection object to follow the execution flow that the gadget chain specifies, the attacker should dynamically set the property `comparator` (line 3) of `PriorityQueue` to an instantiated `TransformingComparator` object, and iteratively sets `TransformingComparator`'s property `transform` (line 16) to another instance `InvokerTransformer` to facilitate the payload `object` in `queue` reaching the sink, as shown in Figure 2b. When this crafted injection object `PriorityQueue` is deserialized via `readObject()`, the payload `object` in `queue` will flow into the security-sensitive call site `Method.invoke()`, thereby allowing remote code execution.
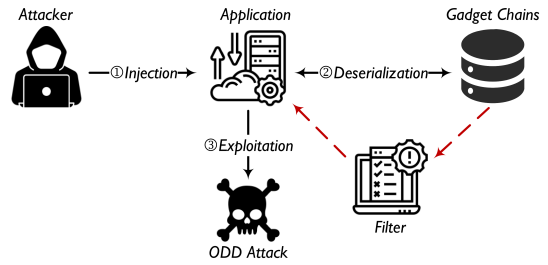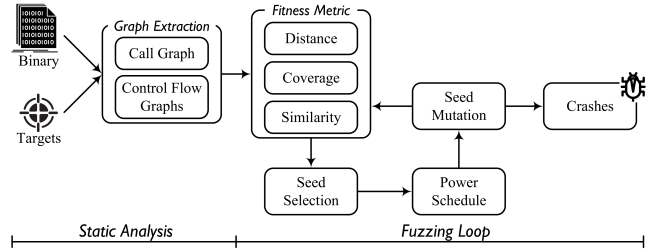
B. Threat Model

Figure 3 illustrates the threat model that we explored in this paper. Assume that there are attacker-controllable *deserialization entry points* in the target Java application. If an attacker ❶ injects crafted objects into these untrusted entry points, the target application will ❷ deserialize these objects and automatically invoke attacker-specified gadget chains on the application's classpath. Then, ❸ malicious payloads carried by injection objects will flow into the security-sensitive call sites, enabling attackers to perform ODD attacks for exploitation.

This assumption is practical because insecure deserialization is common in the Java ecosystem. Take the prevalent commercial platform `Oracle WebLogic Server` (WLS) [28] as an example. The T3 protocol WLS adopts to transport serialized data with other Java programs exposes a large attack surface, which provides untrusted deserialization entry points to attackers for sending payloads to the victim application. Nonetheless, containing deserialization entry points does not mean that the target application is vulnerable. As shown in Figure 3 (red dotted arrows), given that remediating an ODD vulnerability can be particularly difficult and costly, developers prefer adopting whitelists or blacklists to restrict the deserialization of untrusted objects [12], [29], [30]. However, once a new gadget chain is discovered, existing defense solutions can be easily bypassed [31]. Thus, in our threat model, information about whether there are exploitable gadget chains on the application's classpath is more important.

C. Directed Greybox Fuzzing

Greybox Fuzzing (GF) has become an effective method to detect vulnerabilities [15]. With different goals, it can usually be divided into two types: *Coverage-guided Greybox Fuzzing* (CGF) [32]–[34] and *Directed Greybox Fuzzing* (DGF) [35]–[37]. CGF aims to explore previous undiscovered code snip-

TABLE I: State-of-the-art automated gadget chain discovery tools. Intra-TA and Inter-TA respectively represent the intraprocedural and interprocedural taint analysis, while PTA denotes the points-to analysis.

Technique	Static Analysis	Seed Generation	Seed Mutation	Seed Prioritization
GadgetInspector [12]	Intra-TA	-	-	-
SerHybrid [13]	PTA	Heap graph	-	-
FUGIO [18]	Inter-TA	Property tree	Heuristics	Feedback-driven
ODDFUZZ	Intra-TA	Property tree	Step-forward	Target-directed

pets to achieve high code coverage, expecting to accidentally trigger potential vulnerabilities. However, in some scenarios, such as static report verification [38], the vulnerable code is known and demands exploration. Hence, DGF are designed to guide the fuzzer to a specific location of the program to generate a *Proof-of-Concept* (PoC) testcase.

Figure 4 describes the workflow of the directed greybox fuzzing. Typically, DGF can be split into two phases: static analysis and fuzzing loop. At static analysis phase, the directed fuzzer extracts both the call graph and control flow graphs of the program to calculate the inter-procedural distance [35] between the input binary and pre-defined targets. At the fuzzing loop phase, target distance is usually used as feedback information along with other fitness metrics like coverage [39], [40] and similarity [36] to rapidly guide the fuzzer towards the target sites. Then, the directed fuzzer selects the seeds closer to the target sites in the seed pool according to feedback information and allocates proper *energy* (i.e., power scheduling) for mutation. The energy of a seed determines how many new seeds can be generated. Then, the fuzzer adopts various mutation strategies to steer the seeds to evolve towards the desired target sites and executes the instrumented program. A new seed with smaller distance will be preserved for the next fuzzing loop.

III. MOTIVATION

Despite the severe impact of insecure deserialization in practice, existing efforts on automatically discovering Java ODD vulnerabilities (especially exploitable gadget chains) are still unsatisfactory. For example, the state-of-the-art Java gadget chain discovery tool GadgetInspector [12] can only report few exploitable gadget chains in real-world applications. In fact, as we will show in the rest of this section, to achieve both a high recall (identifying more possible gadget chains) and precision (confirming more exploitable gadget chains), a Java ODD vulnerability discovery solution has to tackle three fundamental challenges.

A. Challenge 1: Runtime Polymorphism

The root cause of ODD vulnerabilities is that, the untrusted deserialized objects can *reach* (in terms of control flow) and *affect* (in terms of data flow) the security-sensitive call sites of target applications. Hence, existing works [12], [13] use static analysis to identify a combination of available gadgets

in the code that can be exploited by attackers to customize insecure deserialization paths. However, due to the runtime polymorphism of Java language, virtual method invocations cannot be determined based on the declared types. As a result, it is difficult to precisely infer program paths that would be taken at runtime, resulting in a high false-negative rate.

A straightforward way is to perform *Class Hierarchy Analysis* (CHA) [41] to take a comprehensive view of both explicit and implicit method invocations. Unfortunately, blindly considering *all* available gadgets on the application's classpath will inevitably lead to path explosion because the number of candidate gadgets increases exponentially as length increases. Hence, as shown in Table I, GadgetInspector [12] respectively computes passthrough data flows from method arguments to 1) return values and 2) method invocations, and enumerates all available methods based on the class inheritance hierarchy and method overriding hierarchy to chain exploitable gadgets. However, given that the attacker-controllable property [42] can propagate from a tainted argument to its subclass arguments not tracked, a set of exploitable gadgets will be missed by GadgetInspector since the Java runtime polymorphism is not considered in its intraprocedural taint analysis. FUGIO [18] computes interprocedural data flows on its built depth-bounded call tree to prune infeasible gadget chains. However, when applied to Java ODD gadget chain discovery, this solution may not work because a typical Java application might integrate hundreds of libraries with their own dependencies. This creates a massive pool of classes and methods, making the call tree too deep and breadth to deal with.

B. Challenge 2: Structured Input Construction

To invoke a series of exploitable gadgets on the application's classpath, the structure of injection objects is often organized as a nested form with multilevel sub-objects. Still taking the gadget chain in Figure 1 as an example. To trigger the gadget `compare()` (line 17), the fuzzer should instantiate the class `TransformingComparator` to which the overridden method `compare()` belongs and assign this instance to the field `comparator` (line 3) of the injection object `PriorityQueue` through POP. This brings the challenge to constructing a both syntactically (i.e., the generated injection object can be (de)serialized) and semantically (the generated injection object satisfies certain control- and data-flow constraints that enable the gadget chain) valid fuzzed input, as it requires 1) shaping the injection object's multilevel hierarchy to enable the execution of the reported gadget chain, and 2) assigning proper property values to trigger the security-sensitive call site. Without prior knowledge about such a complex nested form of object structures, traditional fuzzing techniques cannot thoroughly fuzz the entire gadget chain as they hardly figure out complex structures behind each injected object.

An effective solution to handle such nested object structures is generation-based fuzzing techniques. As shown in Figure 5, SerHybrid [13] performs points-to analysis to produce a heap access path (pink-shaded), which satisfies the data-flow constraints of reaching the security-sensitive call site, from
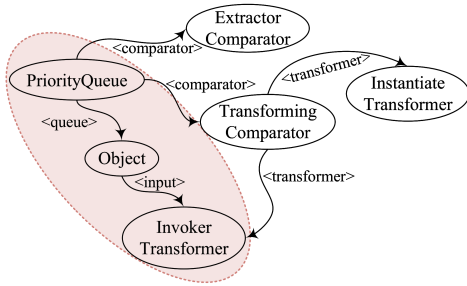
Fig. 5: The heap graph for the gadget chain in Figure 2.

Fig. 6: Overview of ODDFUZZ.

the heap graph and uses fuzzing to assign random values to the field properties not appear in the heap path according to their types to generate valid injection objects for execution. However, as the number of available gadgets increases, a fuzzer unaware to the multilevel class hierarchy of the injection object may hard to assign proper values to those control-data constraints-related properties. For example, it is unlikely to select `TransformingComparator` from a large number of implementations of the interface `Comparator`, resulting in runtime exceptions. FUGIO [18] builds a property tree based on the candidate gadget chain to satisfy the control-flow constraints of sink-reachable injection objects, and mutates each property with some heuristic rules to construct actual injection objects. However, the random combination of arbitrary sub-objects (property values) generated by fuzzing may be semantically (hard to trigger target gadgets) invalid, blocking the injection objects from reaching gadgets closer to target sinks.

C. Challenge 3: Target-Directed Fuzzing

Since the gadget chain is composed of a series of attacker-controllable methods which are automatically executed during object deserialization, the conventional code coverage is not suitable to guide the fuzzer because a generated injection object which triggers more code snippets may not be able to reach the security-sensitive call site of the target chain. For example, an injection object whose property `comparator` is `null` (line 7 in Figure 2a) will be preserved by the coverage-guided fuzzer (e.g., FUGIO) as an interesting seed for the next fuzzing loop since it triggers new code snippet (line 13) in the gadget `siftDown()` (line 6). As a result, the fuzzer will waste most of its time budget on exploring unreachable paths.

Instead of focusing on maximizing the code coverage, DGF prioritizes the seeds whose execution traces are close to the target sites to gain directedness. State-of-the-art directed fuzzers leverage the arithmetic mean of the distances of all the basic blocks on a seed's execution trace to select and schedule the seeds to reach target sites rapidly. However, such a seed distance can be biased and may not entirely correspond to the expected execution path of a gadget chain being validated since not every block drives the seed object to execute towards the target sink expected in an identified chain. Moreover, the execution traces of different seed objects may vary greatly and can only be known at runtime since modifications to a property of the seed object may activate the
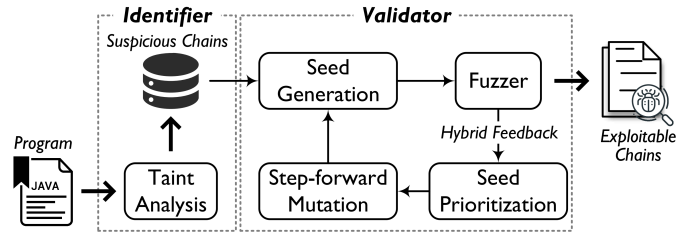
execution of multiple gadgets. Hence, target-directed fuzzing feedback that can effectively evaluate the quality of generated injection objects is desired.

IV. ODDFUZZ DESIGN

To tackle the aforementioned challenges of gadget chain discovery discussed in Section III, we design ODDFUZZ to support structure-aware directed greybox fuzzing. In this section, we outline the overall design and workflow of ODD-FUZZ, and explain its key components.

A. Overview

Figure 6 depicts an overview of ODDFUZZ. The workflow of ODDFUZZ contains two main modules: ❶ In the *identifier* module, ODDFUZZ takes a compiled file (e.g., Jar, War, or Class files) of the *program under testing* (PUT) as input and conducts a lightweight taint analysis to automatically enumerate all potential gadget chains. ❷ In the *validator* module, ODDFUZZ generates a structure-aware seed based on the identified gadget chains to construct syntactically valid injection objects for fuzzing. During the fuzzing loop, ODDFUZZ combines step-forward mutation strategy and hybrid feedback (seed distance and gadget coverage) to guide the fuzzer to mutate injection objects towards the desired sinks. When a generated injection object reaches the security-sensitive call site, the given gadget chain will be reported as an exploitable gadget chain.

B. Taint Analysis

An important condition for constructing an exploitable gadget chain is that whether the attacker-controllable tainted object can propagate from an entry point (i.e., source) to the security-sensitive call site (i.e., sink) method. In other words, if an exploitable gadget chain exists, there must be a call path from the entry point to the security-sensitive call site. A straightforward way is to construct Call Graph (CG) [43], [44] to search for reachable paths [45]. However, due to the Java runtime polymorphism, virtual method invocations cannot be determined based on the declared types. To solve this problem, we perform a lightweight summary-based taint analysis [17], [46], [47] to identify suspicious gadget chains.

Method Summary Computation. ODDFUZZ first computes static summaries for all methods on the classpath of the PUT that are later used for constructing gadget chains.

Specifically, for each method, ODDFUZZ first extracts all its arguments and `this` as method summaries. Then, to track

the information propagation between variables of each method, we focus on four basic statements, including 1) Assign, 2) Load, 3) Store and 4) Call. These statements are widely used for data flow computation in taint analysis. The variable that data-dependent on an argument of the method will also be included in the method's summaries. These method summaries will be used to identify exploitable gadgets whose actual arguments can be controlled by attackers to propagate tainted values by altering the property values of an injection object.

Gadget Chain Identification. Since the gadget chain is a sequence of method invocations that reflects a stack trace from a magic method to a security-sensitive call site, ODD-FUZZ should specify a list of exploitable magic methods and security-sensitive call sites, and identify suspicious gadget chains based on previous computed method summaries. In this paper, we specified a total of 16 magic methods and 30 security-sensitive call sites (as shown in Appendix A).

Then, given that the Breadth-first-search (BFS) adopted by GadgetInspector will skip visited methods (i.e., gadgets which have been traversed on certain infeasible paths will not be considered for gadget chain construction again even if they are exploitable) and thus results in false negatives, once a known magic method is found on the classpath of the PUT, ODDFUZZ performs a Depth-first-search (DFS) starting from this source gadget based on the method summaries to chain exploitable gadgets. To avoid infinite loops (e.g., recursive calls), we set a threshold for the maximum length of candidate gadget chain. Furthermore, to handle the runtime polymorphism of Java language, we perform Class Hierarchy Analysis (CHA) on the call statement only when the caller is tainted, avoiding the path explosion issue caused by blindly considering *all* available gadgets on the application's classpath. In particular, for a call statement $r = x.k(a, \cdots)$, if the caller variable x is tainted (e.g., `comparator.compare()` at line 12 in Figure 1), all overriding methods of method k will be listed as candidates. Otherwise, ODDFUZZ works like a normal CG-based taint analyzer. This iterative analysis procedure will not stop until a security-sensitive sink method is invoked or the maximum length of the enumerated gadget chain exceeds a threshold.

After all paths (i.e., gadget chains) from magic methods to security-sensitive call sites are analyzed, ODDFUZZ runs the validator module for validation. With the help of our lightweight taint analysis, the effectiveness (identifying as many gadget chains as possible) and scalability (analyzing large applications with acceptable time overhead) of ODD-FUZZ in gadget chain identification can be well balanced.

C. Structure-Aware Directed Greybox Fuzzing

Given a target Java application and a candidate gadget chain, ODDFUZZ conducts structure-aware directed greybox fuzzing to generate actual injection objects for validation. The main fuzzing loop is as presented in Algorithm 1 in Appendix B, which is composed of the following three main components.

Structured Seed Generation. As described in Section III-B, constructing a syntactically valid injection object requires 1)
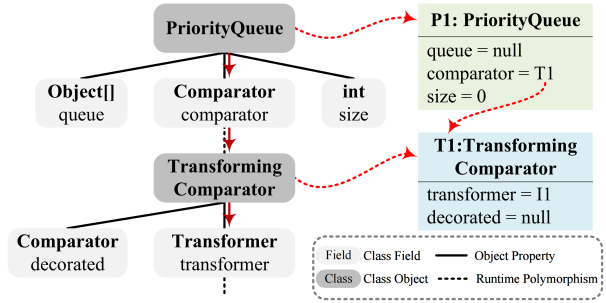

Fig. 7: A merged property tree for injection object generation.

devising its nested object hierarchy that reflects the execution flow of a given gadget chain, and 2) assigning suitable property values to corresponding multilevel sub-objects to facilitate the injection object reaching the sensitive sink. However, heavy use of nested structures makes gadget chain fuzzing ineffective, as it requires well-designed property layout of complex object structures, which is unfriendly to traditional fuzzing solutions.

To this end, we design a structure-aware seed generation approach to handle the complex nested forms by adopting a hierarchical data structure called *property tree* [18], in which the root node represents a class object that holds one or more gadgets, and leaf nodes are a series of class fields which contain the property type and name. As shown in Figure 7, to generate an injection object shown in Figure 2, we instantiate each class involved in the gadget chain and leverage reflection to dynamically collect available properties of each class to construct a property tree. Specifically, if the property type of a field node in a property tree is an object represented (or inherited) by another property tree of which the class holds the next gadget in the target chain, we merge the two property trees by connecting this field node to its corresponding class object node (i.e., the root node of another property tree). It is noteworthy that two property trees are also merged when certain field node's type in a property tree is the interface implemented by the root node (a class object) of another property tree. For example, the property type of the field node `comparator` in the property tree of class `PriorityQueue` is the interface `Comparator`. Hence, the two property trees of class `PriorityQueue` and class `TransformingComparator` are merged by connecting the field node `Comparator comparator` and the root node `TransformingComparator`. We iteratively integrate the property tree based on the invocation order of the gadget chain until there are no more isolated but related sub-trees.

When a suspicious gadget chain is identified by ODDFUZZ, it will be fed into the input generator to construct a corresponding property tree. The multilevel class hierarchy of a target gadget chain can be well modeled with the property tree. Then, the fuzzer starts traversing the backbone of this tree to convert it into an initial injection object for fuzzing (the right side of Figure 7). Other property nodes without successors (e.g., `Object[] queue`) will be set to `null` for mutation.

Seed Prioritization via Hybrid Feedback. Using the above-mentioned structure-aware seed generation that handles the complex nested forms, we can successfully construct syntactically valid injection objects to enable the gadget chain fuzzing process. However, as described in Section III-C, the execution trace of an injection object is dynamically determined, which means that randomly generating and mutating an injection object leads to the sink-unawareness since the property layout of this nested injection object varies greatly in different fuzzing iterations. Such an indeterministic fuzzing campaign without clear feedback guidance degenerates the fuzzer into a semantics-blind dumb fuzzer. As a result, the fuzzer would be confused about which direction to evolve and wastes time on exploring unreachable paths, resulting in low efficiency.

In order to efficiently select and schedule the seeds to reach the security-sensitive call site of a given gadget chain, we propose a hybrid feedback-driven seed prioritization way. Fundamentally, we aim to prioritize and assign more energy to seeds closer to the target security-sensitive call site for mutation. To this end, ODDFUZZ takes two types of feedback metrics into account: *seed distance* and *gadget coverage*.

1) Seed Distance: Computing seed distance to prioritize and schedule seeds to reach target sinks as rapid as possible is a core component of DGF. Following the idea of AFLGo [35] and Hawkeye [36], the distance between a seed s and the target basic block T_b to which the security-sensitive call site belongs is calculated as:

$$d(s, T_b) = \frac{\sum_{m \in \xi(s)} d_b(m, T_b)}{|\xi(s)|} \quad (1)$$

where $d_b(m, T_b)$ is the distance between a basic block m in the execution trace of seed s and the target basic block T_b. It is noteworthy that instead of enumerating all the basic blocks on the execution path of a seed s, we collect the executed basic blocks $\xi(s)$ within the gadgets of the target chain to compute the seed distance, avoiding the fuzzer exploring irrelevant but closer paths.

2) Gadget Coverage: Furthermore, we also adopt gadget coverage (i.e., branch coverage of gadgets in a target chain) as another metric to prioritize seeds which cover more program paths. In the initial fuzzing stage, the gadget coverage aims at guiding the fuzzer to select and prioritize diverse seeds, avoiding getting stuck in local optimum caused by favoring certain seeds with specific execution paths. While in the power scheduling stage, the gadget coverage attempts to give seeds with the same distance but covering more branches higher chances for mutation.

Formally, ODDFUZZ sorts all the generated seeds in ascending order according to their distance and maintains a two-level priority queue. The first seed (or seeds with same distance but different coverage) will be put into the favored queue with higher priority, and the rest of the seeds are put into the less favored queue. Thus, ODDFUZZ has a greater chance to select the next seed from the favored queue for mutation. As for power scheduling, ODDFUZZ uses Equation (2) to consider both seed distance and gadget coverage to assign a proper energy to the selected seed input.

$$p(s, T_b) = \psi(s) \cdot (1 - \widetilde{d}(s, T_b)) \quad (2)$$

where $\psi(s)$ denotes the proportion of the branches of gadgets covered by a seed s (i.e., gadget coverage) to the total branches of all gadgets in a given chain, and $\widetilde{d}(s, T_b) = \frac{d(s, T_b) - minD}{maxD - minD}$ is a normalized seed distance where $minD$ (or $maxD$) is the smallest (or largest) seed distance ever met. It is obvious that $p(s, T_b) \in [0, 1]$ since both the multipliers are in $[0, 1]$.

With Equation (2), the fuzzer can determine the number of mutation chances to be applied on the current seed and evaluate whether the mutated seeds should be favored during the seed prioritization, striking a balance between exploring diverse execution paths and prioritizing a seed that is more likely to reach the desired security-sensitive call site.

Step-Forward Seed Mutation. Previous fuzzing techniques work by randomly mutating binary files via operations like bit flips to produce new inputs. However, such bit-level mutations may lead to invalid syntax when applied to structured inputs. To address this issue, we leverage JQF [20], a parametric fuzzing framework which maps the structured inputs to a sequence of untyped bits (i.e., parameters), to mutate the generated seeds at the bit-level. These bit-level mutations on the parameters correspond to property-level mutations on structured injection objects. Then, ODDFUZZ applies a step-forward seed mutation strategy to efficiently guide the seeds towards the desired security-sensitive call site of a target gadget chain.

Specifically, the fuzzer first traverses the property tree of an injection object to be mutated and checks each property's type. For *primitive* data types (e.g., `boolean`, `int`), the fuzzer uses multiple pseudo-random methods proposed by 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. For example, the parameter sequence for an injection object generated from the property tree in Figure 7 is:

In order to mutate the value of property `size`, which is a variable with `int` type in class `PriorityQueue`, the fuzzer invokes the method `random.nextInt()` to generate a random integer 1. To generate an `Object` array `queue`, the fuzzer invokes the method `random.choose()` to assign it an instance `Object` from the pre-defined dictionary, which is composed of some specific property values (e.g., class object, string object)

involved in *all* classes or methods in the candidate gadget chain. These pre-defined values have a higher probability to satisfy certain hard dependencies during fuzzing.

Furthermore, to guide the seeds towards a desired sink method, ODDFuzz mutates the nested sub-objects of the interesting injection object at the bit-level one by one. To this end, we insert additional `identifier` bytes with the method `random.nextBool()` into the parametric sequence of an injection object. When the fuzzer meets a class object node while traversing the property tree, the fuzzer adds a byte as an identifier to mark whether to mutate the property values of this nested sub-object. We leverage the gadget coverage collected by the fuzzer to identify the class where the last branch covered by the injected object is located. Once an injection object is stuck in certain gadgets, the fuzzer will set the corresponding identifier bytes to *true* and assign random values to parameters, which correspond to structural mutations on the properties of the class to which the stuck gadget belongs, to produce new inputs.

To illustrate our step-forward mutation, considering the following parameter sequence σ_2:

$$\sigma_2 = 0000 \cdots \underbrace{0000\ 0001}_{\text{nextBool()}\rightarrow\text{true}}\ \underbrace{0010\ 0101}_{\text{choose()}\rightarrow\text{?T}} \cdots$$

Identifier   transformer

Suppose that there is an injection object that stops in the gadget `TransformingComparator.compare()`, the fuzzer will flip its `Identifier` to *true* and mutates the parameter sequence (e.g., assigning an instance ?T[2] to the property `transformer`) corresponding to class `TransformingComparator`. Based on this step-forward mutation strategy, the fuzzer can effectively generate semantics-aware inputs which are more likely to reach the target sink.

Finally, when the mutated seed reaches the security-sensitive call site, the fuzzer will report that the given gadget chain is exploitable with the generated injection object.

V. Implementation

We implemented ODDFuzz based on a popular Java fuzzing platform JQF [20]. We customized its components to make it suitable for gadget chain fuzzing while piggybacking on the underlying functionalities of JQF, such as runtime instrumentation.

Taint Analysis. ODDFuzz uses Soot [48] to parse and convert the Java bytecode to the intermediate language Jimple [49]. Based on the basic class information (e.g., class modifier, field, method and instructions) from Jimple, we implemented the method summary-based taint analysis.

Structured Fuzzing. Instead of manually writing declarative specifications of the input format such as context-free grammars or protocol buffers, ODDFuzz modifies the `junit-quickcheck` generators [50] built in JQF to randomly generate and mutate structured injection objects based on the

---

[2]Due to space constraints, we use ?T to represent an instance of *any* class that implements `Transformer`.

candidate gadget chains. To enable and facilitate the structure-aware seed generation, we employ the class `sun.msic.Unsafe` [51] provided by JRE, allowing users to create an instance of a class without invoking its constructor code, initialization code, various JVM security checks and all other low level things.

Runtime Instrumentation. We use the ASM toolkit [52] to instrument Java bytecode on-the-fly via a javaagent as classes are loaded by the JVM. When the PUT starts, the ODDFuzz instrumentor injects a static method invocation that is executed after each call or jump instruction to keep track of the execution trace of an injection object. Note that the instrumentation is limited to gadget chain-related bytecode instead of the whole program for efficiency concerns.

Feedback Collection. For coverage information, we make minimal modifications to JQF to collect branch coverage through instrumenting each basic block based on jump instructions. For distance information, ODDFuzz generates the corresponding intraprocedural control flow graphs (CFGs) of the gadget chain at the bytecode-level based on ASM. The root node (i.e., gadget) of a CFG is identified by the method signature while other CFG nodes are identified by the jump instructions of the corresponding basic blocks. When a gadget chain is fed to the fuzzer for validation, the ODDFuzz distance calculator computes the inter-procedural distance towards the dangerous sink for each basic block based on the invocation order of the gadget chain and generated CFGs. The distance calculator is implemented with JGraphT library [53].

VI. Evaluation

In this section, we evaluate ODDFuzz from different perspectives. First, we measure the effectiveness of ODDFuzz for gadget chain identification, and demonstrate how the structure-aware seed generation and semantic-aware fuzzing guidance of ODDFuzz contribute to triggering exploitable gadgets (Section VI-A). Then, we compare its performance with state-of-the-art automated gadget chain identification tools, including an open-source tool and a previous study (Section VI-B). Finally, we show that ODDFuzz can discover previously unknown vulnerabilities in popular Java applications (Section VI-C).

Experiment Environment. All experiments were conducted on a Linux workstation with an Intel(R) Core(TM) i9-12900k @3.90GHz and 256 GB of RAM, running Ubuntu 18.04.4 LTS with JDK 1.8.0_152.

Benchmark. We performed the evaluation on various gadget chains based on the ysoserial repository [8], a collection of 34 known gadget chains discovered in 22 common Java libraries that can be exploited to perform unsafe object deserialization.

Exploitability Evaluation. To evaluate whether the gadget chains reported by ODDFuzz and baselines were truly exploitable, we employed two professional security analysts to manually inspect each reported gadget chain. For 34 known gadget chains in the benchmark (Section VI-A and VI-B), security analysts compared the gadget chains reported by each approach with the gadget chain attached to the payloads (ground truth) in ysoserial (e.g., as explicitly marked in the

annotation of `CommonsCollections1` [54]). Given that the reported gadget chain may not be completely consistent with the ground truth (e.g., the gadget chain `CommonsCollections1` reported by GadgetInspector and ysoserial), the reported gadget chain would be confirmed as known if its core gadgets[3] involved in the vulnerable application/library were the same to those in ysoserial. For the remaining gadget chains (discovered in ysoserial (Section VI-B) and real-world applications (Section VI-C)), two security analysts manually inspect these reported gadget chains. Once a gadget chain was suspected to be exploitable, they would construct actual exploits for confirmation.

A. Effectiveness

To evaluate the effectiveness of ODDFUZZ, we repeated each experiment 10 times and reported their average statistical performance [55]. We empirically set the threshold for each gadget chain to 15 gadgets. For each statically identified gadget chain, we limit the fuzzing campaign of ODDFUZZ to 120 seconds. We performed additional sensitivity analysis on these two hyperparameters for evaluation in Appendix C.

1) Overall Performance: Table II summarizes the statistics about the evaluation results. The third to fifth columns present the scale of the target applications, including the lines of code (*LoC*), the number of classes, and the number of methods. The sixth and seventh columns show the number of source methods and sink methods covered on an application's classpath. The eighth column represents the number of known gadget chains provided by ysoserial. The *Identified Chains* and *Confirmed Chains* columns respectively mean the number of gadget chains identified by ODDFUZZ's taint analysis module and the number of gadget chains reported by the fuzzing module. Note here that the number in parentheses of the *Identified Chains* columns represents the number of truly exploitable gadget chains in the benchmark identified by the static identifier module, i.e., true positives (TP). For example, ODDFUZZ statically identified nine gadget chains in JDK, two of which are known gadget chains. In the dynamic validation process, these two chains were confirmed by ODDFUZZ with generated injection objects. The last two columns, *Analysis Time* and *Fuzzing Time*, show the total time overhead of taint analysis and fuzzing campaigns in each application.

Overall, in 22 Java libraries, ODDFUZZ statically identified a total of 20 out of 34 known gadget chains, and dynamically generated sink-reachable injection objects for 16 out of these chains without false positives. The results demonstrate the effectiveness of ODDFUZZ in discovering Java ODD gadget chains.

False Positives. In the static analysis stage, we find that among the 583 identified gadget chains, ODDFUZZ correctly discovers 20 known gadget chains with a recall of 58.8% (20/34). In other words, the false-positive rate (FPR) of ODDFUZZ in static analysis is 96.6% (563/583). The root cause is mainly

due to our simple static taint analysis logic. For example, given that some applications implement their own deserialization libraries/protocols (e.g., `XStream` [56] and `Hessian` [57]) instead of using Java native deserialization interfaces (e.g., `Serializable` and `Externalizable`), ODDFUZZ takes all available methods (no matter whether the classes to which they belong inherit the `Serializable` or `Externalizable` interface) on the application's classpath into consideration for gadget chain construction, resulting in a large number of infeasible candidate gadget chains in practice. We discuss this limitation in Section VII and leave the enhancement of static analysis as future effort. Considering that ODDFUZZ has validated these candidate gadget chains through structure-aware directed greybox fuzzing and confirmed 16 known gadget chains from 583 candidates with zero false positives, such a FPR is acceptable.

False Negatives. As shown in Table II, we also find that 14 out of 34 (with a false-negative rate of 41.2%) known gadget chains in ysoserial are missed by ODDFUZZ in static identification stage, mainly due to the limited support for certain dynamic features of Java language such as *reflective calls* [3] and *dynamic proxy* [58]. For example, in `Groovy1` [59], 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 to which classes can be proxied, gadget chains involving dynamic proxy during their construction are difficult to be identified by ODDFUZZ, resulting in false negatives.

In the dynamic verification stage, as shown in Table IV in Appendix, there are four statically identified gadget chains (including `AspectJWeaver` [60], `CommonsCollections1` [54], `CommonsCollections3` [61], and `Jython1` [62]) cannot be dynamically validated by ODDFUZZ. For `AspectJWeaver`, ODDFUZZ fails to generate sink-reachable injection objects because its sink method `writeToPath()` receives a file as input, which cannot be generated by traversing the property tree. For the remaining three gadget chains, their construction involves dynamic proxy[4], which is not supported by our injection object generation and mutation strategy.

2) Impact of Structure-Aware Seed Generation: To construct valid seed objects for gadget chain fuzzing, we proposed a structure-aware seed approach, which leverages the class hierarchy relations between gadgets, to ensure both syntactic and semantic validity of inputs. To evaluate how structure-aware seed contributes to the gadget chain fuzzing of ODDFUZZ, we set up a naive variant of ODDFUZZ, ODDFUZZ-SU (SU: Structure-Unaware), which disables the structure-aware seed. We then reran the experiments 10 times.

The experimental results (the original gadget coverage is reported in Figure 11 in Appendix) are shown in Figure 8, where we can observe that our structure-aware input generator

---

[3]In GadgetInspector, these application-specific continuous gadgets are also called *the building block of the full gadget chain*.

[4]Although ODDFUZZ does not support for dynamic proxy, following GadgetInspector, we regard these known proxy classes as sources to start the gadget chain identification. Hence, these three dynamic proxy-related gadget chains can be statically identified by both ODDFUZZ and GadgetInspector.

TABLE II: Evaluation results of ODDFUZZ on known gadget chains from ysoserial.

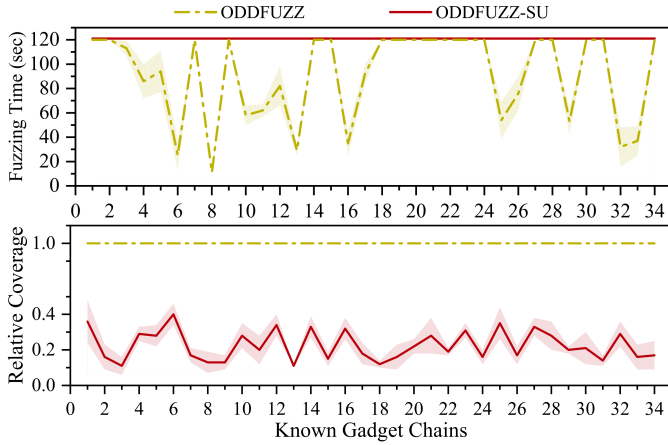| Application	Version	LoC	Classes	Methods	Covered Sources	Covered Sinks	Known Chains	Identified Chains	Confirmed Chains	Analysis Time	Fuzzing Time
JDK	1.7	4.4M	38.5K	324.6K	7	4	4	9 (1)	1	1m51s	16m32s
AspectJWeaver	1.9.2	692.4K	7.1K	19.8K	4	2	1	9 (1)	0	1m56s	18m
BeanShell	2.0b5	44.8K	1.1K	17K	3	1	1	8 (0)	0	1m53s	16m
C3P0	0.9.5.2	30.3K	644	10.1K	6	3	1	13 (1)	1	1m50s	25m53s
Click	2.3.0	10.8K	73	8.5K	4	1	1	8 (1)	1	1m48s	15m26s
Clojure	1.8.0	58.4K	3.8K	25.7K	5	4	1	184 (1)	1	3m30s	6h7m34s
CommonsBeanutils	1.9.2	71.4K	504	7.8K	3	1	1	8 (1)	1	1m52s	14m25s
CommonsCollections	3.1	101K	798	9.7K	7	4	5	97 (5)	3	1m58s	3h10m53s
CommonsCollections4	4.0	101K	630	7.4K	5	2	2	112 (2)	2	1m55s	3h41m9s
FileUpload	1.3.1	10.5K	56	3.1K	3	1	1	8 (0)	0	1m55s	16m
Groovy	2.3.9	252.4K	4.2K	45.6K	4	1	1	13 (0)	0	2m8s	26m
Hibernate	4.3.11	855.7K	7.4K	42.7K	3	1	2	8 (2)	2	2m8s	14m7s
JBossInterceptors	2.0.0	24.2K	166	2.3K	2	1	1	8 (0)	0	1m51s	16m
JSON	2.4	28K	172	5.9K	3	2	1	9 (0)	0	1m52s	18m
JavassistWeld	3.12.1	60.4K	813	11.3K	2	1	1	8 (0)	0	1m58s	16m
Jython	2.5.2	271.9K	6.7K	66.4K	4	1	1	32 (1)	0	2m54s	1h4m
MozillaRhino	1.7R2	118.7K	329	8.2K	4	2	2	7 (2)	2	1m56s	12m10s
Myfaces	2.2.9	330.1K	1.8K	22.8K	2	1	2	7 (0)	0	2m1s	14m
ROME	1.0	94.5K	423	6.9K	2	1	1	5 (1)	1	1m48s	8m53s
Spring	4.1.4	904.3K	1.3K	14.5K	3	2	2	10 (0)	0	1m59s	20m
Vaadin	7.7.14	572.1K	4.5K	17.5K	4	1	1	13 (1)	1	1m54s	24m37s
Wicket	6.23.0	420.7K	3.2K	11.1K	2	1	1	7 (0)	0	1m50s	14m
Total	-	-	-	-	-	-	34	583 (20)	16	-	-

Fig. 8: Comparison of ODDFUZZ and ODDFUZZ-SU. The x-axis is 34 known gadget chains listed in Table IV. The y-axes are the fuzzing time and relative coverage (i.e., the ratio of gadget coverage that obtained by ODDFUZZ-SU and ODDFUZZ), respectively.

Fig. 9: Coverage comparison of ODDFUZZ, ODDFUZZ-RM, ODDFUZZ-CG and ODDFUZZ-DG. We use the results of ODDFUZZ as the baseline. The x-axis is 34 known gadget chains from ysoserial. The y-axis is the relative coverage of each variant against ODDFUZZ.

can effectively construct syntactically and semantically valid seed objects, successfully validating the target chain within the given time budget. In some cases (e.g., `CommonsCollections2` in `Apache Commons Collections4` library), ODDFUZZ takes only a dozen seconds to validate the target gadget chain. By contrast, ODDFUZZ-SU is unable to validate any reported gadget chain. That is because structure-unaware fuzzing has no prior knowledge of object structure provided by target gadget chains, thus stuck in the initial fuzzing stage. As shown in Figure 8, the lack of structured injection objects makes the fuzzer only trigger a few gadgets. This result demonstrates the effectiveness of structure-awareness in gadget chain fuzzing, which allows us to achieve performance improvement in
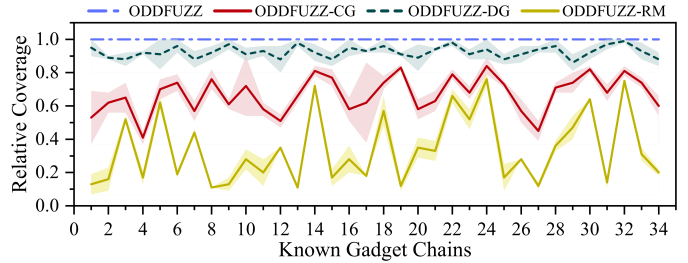
dynamic validation.

3) Impact of Feedback-Driven Fuzzing Guidance: To effectively guide the injection objects to evolve towards desired gadgets, we proposed a feedback-driven fuzzing guidance that is armed with two key strategies, i.e., step-forward mutation and hybrid feedback-based seed prioritization. To evaluate how step-forward mutation and hybrid feedback contribute to the gadget chain fuzzing of ODDFUZZ, we also set up the variants of ODDFUZZ, ODDFUZZ-RM (RM: Random Mutation), ODDFUZZ-DG (DG: Distance-Guided) and ODD-FUZZ-CG (CG: Coverage-Guided), which disables the step-forward mutation and decouples the distance feedback and coverage feedback, respectively. We then reran the experiments 10 times again.

The experimental results (the original gadget coverage is reported in Figure 12 in Appendix) are shown in Figure 9, where we can observe that ODDFUZZ discovers more valid branches than ODDFUZZ-RM, ODDFUZZ-DG and ODD-FUZZ-CG in almost every gadget chain, guiding the fuzzer

to evolve towards desired gadgets. For instance, in 25 of 34 gadget chains (73.5%), the random mutation-based fuzzer ODDFUZZ-RM only discovers branches that are less than half of ODDFUZZ. According to our analysis, it is mainly because blind mutation to object properties makes the fuzzer continue to explore shallow gadgets or infeasible paths, limiting the capability of the fuzzer to preserve critical waypoints during fuzzing. In addition, we also observe that compared to coverage-guided fuzzing, distance-guided fuzzing significantly increases the valid branches triggered by the fuzzer (the improvement is more than 40% in certain cases). This result demonstrates the effectiveness of both strategies, step-forward mutation and hybrid feedback, as both of them contribute to the directedness of gadget chain fuzzing, and their combination allows ODDFUZZ to carry the fuzzing exploration towards the desired sinks.

B. Comparison with State-of-the-Art Work

We compared ODDFUZZ with two state-of-the-art automated gadget chain discovery tools, GadgetInspector [12] and SerHybrid [13]. As shown in Table III (detailed results[5] are reported in Table IV in Appendix). The *Identified Chains* column represents the number of potential gadget chains statically identified by each approach, and the *Confirmed Chains* column indicates how many of them are disclosed as exploitable in ysoserial. Considering that GadgetInspector is purely static, its *Confirmed Chains* column denotes the static results, while in SerHybrid and ODDFUZZ, the *Confirmed Chains* column shows the results after dynamic validation. Similar to Table II, the *Analysis Time* and *Fuzzing Time* columns respectively represent the total time cost of static analysis and fuzzing campaigns in each approach. Note that, as reported in [13], 13 out of 22 applications (involving 19 exploitable gadget chains) are not evaluated by SerHybrid because it specifically focuses on reflection-enabled Java ODD vulnerabilities (labeled as "N/A"), and two applications (Clojure and Jython) cannot be analyzed statically within given time budgets (labeled as "Timeout").

Overall, ODDFUZZ achieves significant performance improvement in all applications. In particular, ODDFUZZ reported 16 out of 34 exploitable gadget chains without false positives, including 13 unique gadget chains that cannot be found by baselines. By contrast, the number of truly exploitable gadget chains reported by GadgetInspector and SerHybrid is three and two, respectively.

ODDFUZZ vs. GadgetInspector. As shown in Table III, GadgetInspector takes an average of 41 seconds to analyze each application and reports 116 suspicious gadget chains. However, only three of them are exploitable, meaning that 97.4% of them are false positives. Such a significant performance gap mainly results from two aspects. On the one

hand, constrained by a few simplifying assumptions (e.g., all members of a tainted object are also tainted) and requiring manual inspection of the reports, GadgetInspector is prone to precision issues and cannot guarantee that identified gadget chains are truly exploitable. On the other hand, due to the lack of consideration of Java runtime polymorphism when computing intraprocedural data flows, GadgetInspector suffers from unsound analysis results, resulting in missed available gadgets, i.e., false negatives. By contrast, as reported in Table IV, benefiting from our lightweight summary-based taint analysis, ODDFUZZ statically identifies 17 more exploitable gadget chains (covering all three chains reported by GadgetInspector) and dynamically validated 15 out of them with no false positives. It is noteworthy that CommonsCollections1 [54], which can be identified by GadgetInspector, fails to be validated by our approach because of certain specific cases (e.g., dynamic proxy discussed in Section VI-A) in injection object construction. Nevertheless, the detection capability of ODDFUZZ is still promising (significantly improving the recall rate of static analysis with acceptable time overhead) and these limitations can be solved to some extent (as discussed in Section VII).

ODDFUZZ vs. SerHybrid. As reported in Table III, SerHybrid successfully confirms two exploitable gadget chains in under two minutes on average. Despite its promising performance, a major drawback of SerHybrid lies in that most (32 out of 34) exploitable gadget chains are missed. For instance, although SerHybrid statically identifies three potential gadget chains in Hibernate in under one hour, it cannot generate an injection object for any of them for validation within 30 minutes (the time budget set by [13]). According to our manual analysis, it is mainly because an execution path from a source object to the sink object with a tainted flow cannot provide the fuzzer with class hierarchy information required for injection object generation. By contrast, owing to our structure-aware directed greybox fuzzing, ODDFUZZ efficiently generates both syntactically and semantically valid injection objects for 16 exploitable gadget chains (including two chains in Hibernate) within two minutes on average.

C. Vulnerability Discovery

We chose target applications that satisfied the following criteria. First, they are Java projects since ODDFUZZ is designed to search deserialization vulnerabilities in Java. Second, as discussed in our threat model, these applications should contain known deserialization entries because we prefer to actually exploit the Java deserialization vulnerabilities with found gadget chains rather than just discover potential chains. Third, they cover diverse application domains so that the generality of our approach can be evaluated. Using the three criteria, we selected four target Java applications, including Oracle WebLogic Server (a commercial application server), Sonatype Nexus (a repository manager), Apache Dubbo (a high-performance Remote Procedure Call (RPC) framework), and protostuff (a Java serialization library), to demonstrate

---

[5]Unfortunately, despite our best effort, the implementation of SerHybrid was not reproducible. We made unsuccessful attempts to contact the authors for suggestions. Hence, we compare them against the results published in their paper. To ensure fairness, we carefully make the experimental settings and only compare them with the same libraries they tested.

Application	Known Chains	GadgetInspector			SerHybrid			ODDFUZZ			
		Identified Chains	Confirmed Chains	Analysis Time	Identified Chains	Confirmed Chains	Analysis Time	Identified Chains	Confirmed Chains	Analysis Time	Fuzzing Time
JDK	4	5	0	53s	N/A	N/A	N/A	9 (1)	1	1m51s	16m32s
AspectJWeaver	1	6	0	41s	N/A	N/A	N/A	9 (1)	0	1m56s	18m
BeanShell	1	2	0	49s	1	0	10m55s	8 (0)	0	1m53s	16m
C3P0	1	2	0	48s	N/A	N/A	N/A	13 (1)	1	1m50s	25m53s
Click	1	4	0	39s	N/A	N/A	N/A	8 (1)	1	1m48s	15m26s
Clojure	1	12	1	40s	N/A	N/A	Timeout	184 (1)	1	3m30s	6h7m34s
CommonsBeanutils	1	2	0	37s	0	0	13m6s	8 (1)	1	1m52s	14m25s
CommonsCollections	5	4	1	39s	1	1	26m51s	97 (5)	3	1m58s	3h10m53s
CommonsCollections4	2	4	0	38s	1	1	11m21s	112 (2)	2	1m55s	3h41m9s
FileUpload	1	3	0	38s	N/A	N/A	N/A	8 (0)	0	1m55s	16m
Groovy	1	4	0	47s	3	0	1h26m	13 (0)	0	2m8s	26m
Hibernate	2	3	0	41s	3	0	56m37s	8 (2)	2	2m8s	14m7s
JBossInterceptors	1	2	0	38s	N/A	N/A	N/A	8 (0)	0	1m51s	16m
JSON	1	2	0	39s	N/A	N/A	N/A	9 (0)	0	1m52s	18m
JavassistWeld	1	2	0	39s	N/A	N/A	N/A	8 (0)	0	1m58s	16m
Jython	1	42	1	50s	N/A	N/A	Timeout	32 (1)	0	2m54s	1h4m
MozillaRhino	2	3	0	40s	N/A	N/A	N/A	7 (2)	2	1m56s	12m10s
Myfaces	2	2	0	37s	N/A	N/A	N/A	7 (0)	0	2m1s	14m
ROME	1	2	0	36s	0	0	6m30s	5 (1)	1	1m48s	8m53s
Spring	2	2	0	38s	N/A	N/A	N/A	10 (0)	0	1m59s	20m
Vaadin	1	5	0	37s	N/A	N/A	N/A	13 (1)	1	1m54s	24m37s
Wicket	1	3	0	36s	N/A	N/A	N/A	7 (0)	0	1m50s	14m
Total	34	116	3	-	9	2	-	583 (20)	16	-	-

the vulnerability discovery capability of ODDFUZZ in practical scenarios.

1) Unknown Vulnerability Discovery: An overview of the vulnerabilities found by ODDFUZZ is shown in Table V in Appendix. In total, ODDFUZZ has successfully detected six previously unknown Java ODD vulnerabilities. While three of them were found within `Oracle WebLogic Server`, the remaining three vulnerabilities respectively arose from `Sonatype Nexus`, `Apache Dubbo`, and `protostuff`. These vulnerabilities can be exploited to perform RCE attacks by building our newly discovered gadget chains. We have responsibly reported all the vulnerabilities to corresponding vendors and have received their positive feedback. At the time of paper writing, five of the vulnerabilities have been patched and assigned CVE numbers due to their severe security consequences.

2) Case Study: ODDFUZZ uncovered a RCE vulnerability (CVE-2020-14756 [63]) in the `Oracle Coherence` product of `Oracle WebLogic Server`. Successful attack of this vulnerability can result in takeover of `Oracle Coherence`. As shown in Figure 13a, the flow of triggering the vulnerability is as follows: ❶ The attacker first instantiates `PriorityQueue` to reuse the known entry gadget (magic method) `readObject()` which unconditionally invokes the second to fourth gadget (line 5-7). ❷ To connect the fifth gadget `ExtractComparator.compare()`, the field `comparator` of class `PriorityQueue` should be set to the `ExtractorComparator`'s instance through POP. ❸ Following the above steps to recursively modify the injection object's properties to trigger the security-sensitive call site `Method.invoke()` (line 32). The complete gadget chain of CVE-2020-14756 is shown in Figure 13b in Appendix.

It is difficult to validate this exploitable gadget chain by traditional fuzzing, as it requires prior knowledge about the layout of the multilevel sub-objects to avoid the fuzzing campaign stuck in the initial stage due to randomly generated

property values. However, ODDFUZZ is able to produce a syntactically valid injection object to facilitate gadget chain execution by supplying the fuzzed structure extracted from the nested hierarchy of multiple gadget classes to the input generator. In this way, the injection object can be fuzzed to reach the sink. Moreover, in the fuzzing campaign, ODDFUZZ mutated the fuzzed object step by step towards the sink and finally triggered the vulnerability.

VII. DISCUSSION

Better Static Analysis. As discussed before, our static analysis suffers from precision and soundness problems. Specifically, false positives can be introduced by our simple taint analysis logic. For example, ODDFUZZ does not restrict the candidate space of available gadgets according to application-specific deserialization libraries/interfaces and not handle several special cases (e.g., an untrusted variable modified by the keyword `transient` cannot be deserialized). These false positives can be reduced by improving taint analysis rules. By contrast, false negatives can be introduced by missing indirect call targets caused by certain dynamic features (reflective calls, dynamic proxy, etc.). Fortunately, benefiting from recent solutions [3], [58] which can (partially) solve these advanced language features, the unsoundness of our approach can be well mitigated. In addition, similar to existing works [12], [13], [18], [24], the effectiveness (recall) of our static gadget chain identification also relies heavily on the prior expert knowledge of available sources and sinks, which is the main manual effort required in ODDFUZZ. Considering that there are a few orthogonal tools/approaches [11], [64] 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 ODD vulnerabilities in the wild can be improved.

Diverse Generation Strategy. As evaluated in Section VI-A, the awareness of object structure can help improve the performance in gadget chain fuzzing. However, the optimal generation strategy needs to be suitable for diverse deserialization scenarios and might be changed depending on the construction of a gadget chain. For example, the exploitation of certain available gadgets relies on some specific techniques (e.g., the dynamic proxy used in `Groovy1` [59]) or constraints (e.g., the file input required by `AspectJWeaver` [60]), which blocks the injection object generation with the property tree. A possible solution is to design some general templates for these specifications. We leave such exploration for our future work.

Exploit Construction. ODDFUZZ also requires some human efforts to help construct practical exploits because the injection objects constructed by our structure-aware generator represent the minimum effort required to trigger the gadget chains. In order to construct actual exploits, security analysts need to further ❶ check whether the tainted properties flowing into the security-sensitive call site are attacker-controllable. If it is truly attacker-controllable, security analysts should ❷ manually replace the non-harmful command (e.g., open calculator) with a malicious one (e.g., reverse shell) based on the injection objects generated by ODDFUZZ. We intend to find a more automatic way as future work, while in this work, the core goal of ODDFUZZ is to efficiently discover exploitable Java ODD gadget chains from a large number of static analysis reports.

VIII. RELATED WORK

A. Deserialization Vulnerabilities in Java

Vulnerability Mitigation. Most existing works focus on understanding and protecting applications against known deserialization vulnerabilities [65]. Muñoz et al. [66] conducted a comprehensive analysis on JSON deserialization libraries and presented several mitigation measures as takeaways. Carettoni [29] presented a configurable Java deserialization library, which supports multiple optional settings such as blacklist and whitelist, to secure application from untrusted input. Cristalli et al. [67] designed a novel sandbox system, which collects the behavior information of benign deserialization process and constructs the precise execution path, to mitigate the problems of deserialization of untrusted data in Java.

Vulnerability Detection. Despite the existing efforts, these defense solutions will be bypassed once a new gadget or fundamental vector is found [31]. Hence, some works focus on detecting potential vulnerabilities in applications [68]–[70]. Koutroumpouchos et al. [64] proposed an extendable tool *ObjectMap* which generates a series of requests to detect whether the payload can be directly passed to the target application. Similarly, Marshalsec [9] and Java Deserialization Scanner [11] are two tools that dynamically scan and exploit know gadget chains from the ysoserial project [8].

Automated Gadget Chain Discovery. In order to automatically identify new gadget chains, Haken [12] presented Gad-getInspector, 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 attacker-controllable gadget chains. Chen et al. [71] developed Tabby, a graph-based static analysis tool, to support gadget chain discovery. Rasheed et al. [13] proposed SerHybrid, a hybrid analysis-based approach which constructs a heap abstraction to produce actual injection objects to automatically validate exploitable gadget chains. Cao et al. [72] proposed GCMiner, which captures both explicit and implicit method calls to identify candidate gadget chains, and adopts an overriding-guided object generation approach to guarantee the validity of injection objects during fuzzing. By contrast, ODDFUZZ aims to improve the effectiveness and efficiency of gadget chain validation via structure-aware directed greybox fuzzing.

B. Deserialization Vulnerabilities in Other Languages

Security threats of insecure deserialization also exist in other mainstream programming languages [23]–[25]. Dahse et al. [24], [73] conducted static taint analysis to detect gadget chains in common PHP applications. FUGIO [18] combined coarse-grained program analysis and fuzzing to automatically produce exploit objects for PHP Object Injection (POI) vulnerabilities. Shahriar and Haddad [74] proposed a lightweight approach based on latent semantic indexing to identify Object Injection Vulnerabilities (OIVs) in web application. They identified multiple keywords that are likely responsible for OIVs and defined customized queries to identify relevant source files to discover new vulnerabilities. SerialDetector [25] studied the root cause of OIVs in .NET applications and presented a scalable taint-based data flow analysis to discover and leverage publicly available gadgets.

IX. CONCLUSION

In this paper, we propose a novel hybrid solution ODD-FUZZ to efficiently discover Java deserialization vulnerabilities. ODDFUZZ performs lightweight static taint analysis to identify candidate *gadget chains* and applies a structure-aware directed fuzzing to mitigate false positives. Results show that, ODDFUZZ could discover 16 out of 34 known gadget chains, while two state-of-the-art baselines only identify three of them. Moreover, we have discovered six previously unreported exploitable gadget chains and five of them have been assigned with CVE-IDs.

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. 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.

[4] 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.

[5] 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.

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

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

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

[9] Marshalsec, 2022, https://github.com/mbechler/marshalsec.

[10] Joogle, 2022, https://github.com/Contrast-Security-OSS/joogle.

[11] Java Deserialization Scanner, 2022, https://github.com/federicodotta/Java-Deserialization-Scanner.

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

[13] 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.

[14] 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.

[15] 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.

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

[17] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. L. Traon, D. Octeau, and P. D. McDaniel, "Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," in *Proceedings of the 35th Annual ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2014, pp. 259–269.

[18] 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.

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

[20] 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.

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

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

[23] S. Lekies, K. Kotowicz, S. Groß, E. A. V. Nava, and M. Johns, "Code-reuse attacks for the web: Breaking cross-site scripting mitigations via script gadgets," in *Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2017, pp. 1709–1723.

[24] 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.

[25] 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.

[26] 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 23rd ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2016, pp. 779–790.

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

[28] WebLogic, 2022, https://www.oracle.com/security-alerts/.

[29] Carettoni, "Defending against java deserialization vulnerabilities," 2016.

[30] JEP290, 2022, http://openjdk.java.net/jeps/290.

[31] Y. Zhang, Y. Wang, K. Li, and K. Chai, "New exploit technique in java deserialization attack," in *Proceedings of the Black Hat EU*, 2019.

[32] M. Böhme, V. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," in *Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2016, pp. 1032–1043.

[33] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," in *Proceedings of the 39th IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 711–725.

[34] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen, "Collafl: Path sensitive fuzzing," in *Proceedings of the 39th IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 679–696.

[35] M. Böhme, V. Pham, M. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2017, pp. 2329–2344.

[36] H. Chen, Y. Xue, Y. Li, B. Chen, X. Xie, X. Wu, and Y. Liu, "Hawkeye: Towards a desired directed grey-box fuzzer," in *Proceedings of the 25th ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2018, pp. 2095–2108.

[37] H. Huang, Y. Guo, Q. Shi, P. Yao, R. Wu, and C. Zhang, "Beacon: Directed grey-box fuzzing with provable path pruning," in *Proceedings of the 43rd IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022.

[38] M. Christakis, P. Müller, and V. Wüstholz, "Guiding dynamic symbolic execution toward unverified program executions," in *Proceedings of the 38th International Conference on Software Engineering (ICSE)*. ACM, 2016, pp. 144–155.

[39] C. Aschermann, S. Schumilo, A. Abbasi, and T. Holz, "Ijon: Exploring deep state spaces via fuzzing," in *Proceedings of the 41st IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 1597–1612.

[40] H. Wang, X. Xie, Y. Li, C. Wen, Y. Li, Y. Liu, S. Qin, H. Chen, and Y. Sui, "Typestate-guided fuzzer for discovering use-after-free vulnerabilities," in *Proceedings of the 42nd International Conference on Software Engineering (ICSE)*. ACM, 2020, pp. 999–1010.

[41] 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)*, vol. 952. Springer, 1995, pp. 77–101.

[42] 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.

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

[44] 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 International Conference on Software Engineering (ICSE)*. ACM, 2022, pp. 1456–1468.

[45] 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.

[46] S. Arzt and E. Bodden, "Stubdroid: automatic inference of precise data-flow summaries for the android framework," in *Proceedings of the 38th International Conference on Software Engineering (ICSE)*. ACM, 2016, pp. 725–735.

[47] H. Tang, X. Wang, L. Zhang, B. Xie, L. Zhang, and H. Mei, "Summary-based context-sensitive data-dependence analysis in presence of callbacks," in *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 2015, pp. 83–95.

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

[49] R. Vallee-Rai and L. J. Hendren, "Jimple: Simplifying java bytecode for analyses and transformations," 1998.

[50] P. Holser, "junit-quickcheck: Property-based testing, junit-style," 2014.

[51] The Unsafe Class: Unsafe at Any Speed, 2022, https://blogs.oracle.com/javamagazine/post/the-unsafe-class-unsafe-at-any-speed.

[52] ASM, 2022, https://asm.ow2.io.

[53] JGraphT, 2022, hhttps://jgrapht.org/.

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

[55] 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.

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

[57] Hessian, http://hessian.caucho.com/doc/hessian-serialization.html, 2022.

[58] 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.

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

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

[61] CommonsCollections3, 2022, https://github.com/frohoff/ysoserial/blob/master/src/main/java/ysoserial/payloads/CommonsCollections3.java.

[62] Jython1, 2022, https://github.com/frohoff/ysoserial/blob/master/src/main/java/ysoserial/payloads/Jython1.java.

[63] CVE-2020-14756, 2022, https://www.oracle.com/security-alerts/cpujan2021.html.

[64] 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.

[65] 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.

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

[67] 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.

[68] 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.

[69] 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.

[70] 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.

[71] 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.

[72] S. Cao, X. Sun, X. Wu, L. Bo, B. Li, R. Wu, W. Liu, B. He, Y. Ouyang, and J. Li, "Improving java deserialization gadget chain mining via overriding-guided object generation," in *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering (ICSE)*. IEEE, 2023.

[73] J. Dahse and T. Holz, "Simulation of built-in PHP features for precise static code analysis," in *Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2014.

[74] H. Shahriar and H. Haddad, "Object injection vulnerability discovery based on latent semantic indexing," in *Proceedings of the 31st Annual ACM Symposium on Applied Computing (SAC)*. ACM, 2016, pp. 801–807.

APPENDIX A
TARGET SOURCES AND SINKS

Magic methods (sources) and security-sensitive call sites (sinks) covered by ODDFuzz are listed below, among which six sources and 16 sinks are considered by GadgetInspector (highlighted in gray). These sensitive call sites can be exploited to perform *Remote Code Execution* (RCE), *JNDI injection* (JNDIi), *System Resource Access* (SRA), and *Server-Side Request Forgery* (SSRF) attacks. It is noteworthy that, similar to GadgetInspector, we explicitly maintain a set of magic methods (security-sensitive call sites) of specific classes as sources (or sinks) for gadget chain identification because not all of them can be exploited (e.g., in Clojure, only `clojure.main$eval_opt.invoke()` instead of `clojure.core$comp$fn_4727.invoke()` is vulnerable).

- **Magic Methods:** `readObject`, `hashCode`, `get`, `put`, `compare`, `readExternal`, `readResolve`, `finalize`, `equals`, `compareTo`, `toString`, `validateObject`, `readObjectNoData`, `<clinit>`, `call`, `doCall`

- **Security-Sensitive Call Sites:**
  - *Remote Code Execution* (**RCE**): `getDeclaredMethod`, `getConstructor`, `findClass`, `getMethod`, `loadClass`, `start`, `exec`, `invoke`, `forName`, `newInstance`, `exit`, `defineClass`, `call`, `invokeMethod`, `invokeStaticMethod`, `invokeConstructor`
  - *JNDI Injection* (**JNDIi**): `getConnection`, `do_lookup`, `lookup`, `c_lookup`, `getObjectInstance`, `connect`
  - *System Resource Access* (**SRA**): `newBufferedReader`, `newBufferedWriter`, `delete`, `newInputStream`, `newOutputStream`, `<init>`
  - *Server-Side Request Forgery* (**SSRF**): `openConnection`, `openStream`

APPENDIX B
MAIN FUZZING LOOP OF ODDFUZZ

Algorithm 1 Gadget Chain Fuzzing

Input: the chain to be validated c
Output: sink-triggering seed set $\mathcal{S}_{\times}$

1: $\mathcal{S}_{\times} \leftarrow \emptyset$
2: $\mathcal{S} \leftarrow$ GENERATEINITIALSEED(c)
3: $minDistance \leftarrow \infty$
4: $gadgetCoverage \leftarrow \emptyset$
5: **repeat**
6:     $s \leftarrow$ SELECTSEED($\mathcal{S}$)
7:     $p \leftarrow$ ASSIGNENERGY(s)
8:     **for** i from 1 to p **do**
9:         $s' \leftarrow$ MUTATESEED(s)
10:         $result \leftarrow$ EXECUTEPROGRAM(s')
11:         **if** REACHSINK($result$) == *true* **then**
12:             $\mathcal{S}_{\times} \leftarrow \mathcal{S}_{\times} \cup s'$
13:             EMITSIGNAL(c, "Reachable")
14:         **else if** $s'.distance < minDistance$ **then**
15:             $\mathcal{S} \leftarrow \mathcal{S} \cup s'$
16:             $minDistance \leftarrow s'.distance$
17:         **else if** $s'.coverage \nsubseteq gadgetCoverage$ **then**
18:             $\mathcal{S} \leftarrow \mathcal{S} \cup s'$
19:             $gadgetCoverage \leftarrow s'.coverage$
20:         **end if**
21:     **end for**
22: **until** *timeout* or *sink-triggering* signal received

Algorithm 1 describes the overall process of our gadget chain fuzzing. Given a target Java application and an identified gadget chain, the fuzzer initiates a fuzzing campaign during a
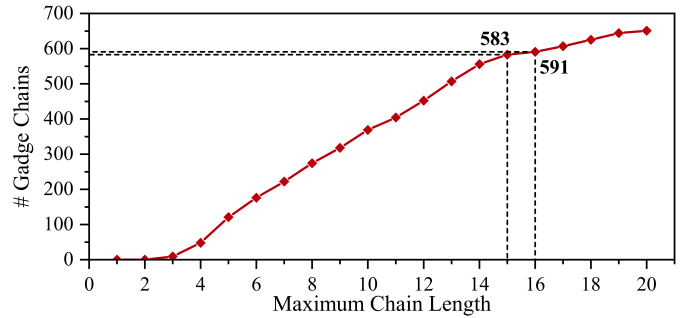
given time budget. The fuzzing process starts by adding the initial injection object generated from the candidate gadget chain c to the prepared seed pool S (line 1-2) and initializing feedback information (line 3-4). It then repeats the following fuzzing loops until finding an injection object that can reach the security-sensitive call site in the gadget chain. To schedule favored seeds for fuzzing in each round, the fuzzer selects a set of seeds s with higher priority from the seed pool S based on their previous execution feedback (line 6). Each chosen seed is assigned to a certain amount of power that determines how many new seed inputs can be derived in this round (line 7). Next, the fuzzer mutates the scheduled seed to generate a new injection object s' to execute the instrumented program (line 9-10). When this mutated seed s' reaches the security-sensitive call site, the fuzzer adds this seed to the sink-triggering seed set $S_\times$ and emits a signal to ODDFUZZ to stop the fuzzing campaign of the target gadget chain (line 11-13). If this mutated seed does not reach the sink but contributes to reducing the seed distance towards the target sink, the fuzzer derives new seeds and updates the current minimal seed distance *minDistance* (line 14-16). Furthermore, if this mutated seed executes more branches within gadgets on the execution path of the target chain, the fuzzer also adds this seed to the seed pool and updates the current gadget coverage *gadgetCoverage* (line 17-19). The fuzzing loop will not stop until the given time budget expires or sink-triggering signal is received by ODDFUZZ (line 22). The remaining section details each step in the fuzzing process.
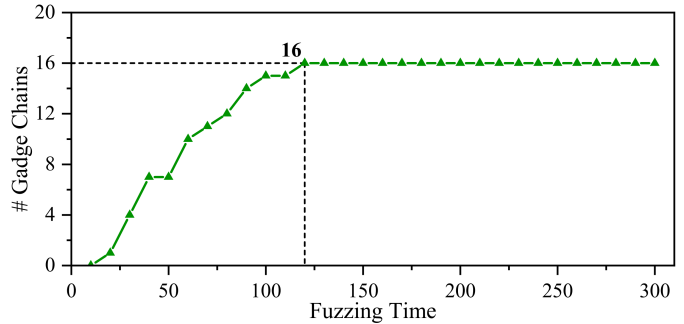
APPENDIX C
HYPERPARAMETER EVALUATION

To evaluate the optimal setting of two hyperparameters, including the maximum length of gadget chains (*Gadget Chain Length*) that ODDFUZZ analyzes and the time budget assigned to fuzzing each candidate gadget chain (*Fuzzing Time Budget*), we conducted the following experiments.

Gadget Chain Length. Gadget chain length denotes the maximum quantity of gadgets that ODDFUZZ could chain in our static taint analysis. It is one of the critical hyperparameters in our approach because it determines the recall of gadget chain identification. For evaluation, we singly run the static identifier module of ODDFUZZ on ysoserial and counted the number of identified gadget chains within a specified maximum length, from 1 to 20 in increment of 1. For example, if the maximum threshold was set to 2, we would count the number of gadget chains with length of 1 and 2. According to the assessment of our employed security experts, it is reasonable to set the value range as $[1, 20]$ because the length of most publicly disclosed gadget chains is less than 20.

The evaluation results are shown in Figure 10a. We can find that the growth rate of the number of newly discovered gadget chains slows down (less than 10 chains for the first time) when the maximum gadget chain length is raised from 15 to 16. Therefore, we set the maximum gadget chain length to be 15 in Section VI.

(a) Gadget chain length.

(b) Fuzzing time budget.

Fig. 10: Sensitivity analysis on gadget chain length and fuzzing time budget.

Fuzzing Time Budget. Fuzzing time budget represents the maximum time budget assigned to the fuzzer for validating a candidate gadget chain. It is another critical hyperparameter in our approach because assigning too much time to fuzz a gadget chain unable to be exploited will waste computation resources. Similar to the aforementioned experiment involving maximum gadget chain length, we used the 34 known gadget chains in ysoserial for evaluation. For each gadget chain, we ran the fuzzer 10 times and counted the number of validated gadget chains within a specified maximum length, from 0 to 300 seconds in increments of 10 seconds [18]. As reported in Figure 10b, ODDFUZZ cannot successfully validate more gadgets after 120 seconds (2 minutes). Hence, we set the fuzzing time budget to be 120 seconds in all experiments in Section VI.
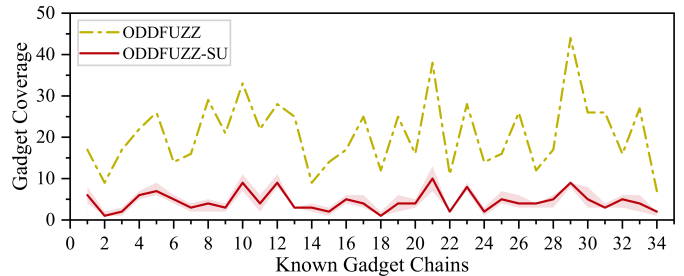

Fig. 11: Gadget coverage comparison of ODDFUZZ and ODDFUZZ-RM. The x-axis is 34 known gadget chains listed in Table IV and the y-axis is the gadget coverage.
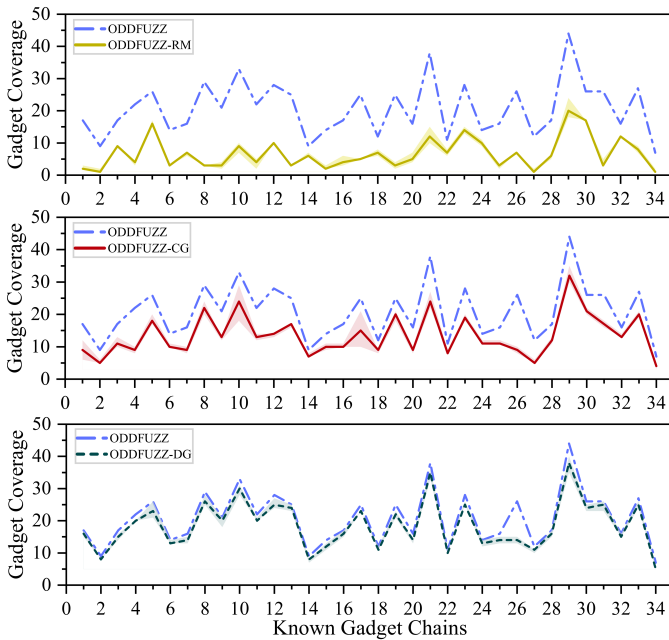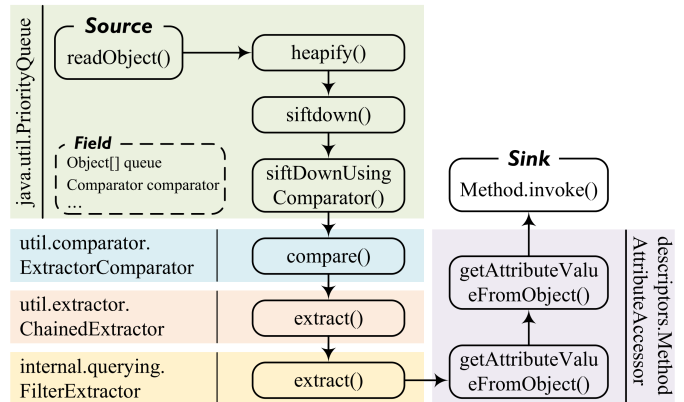
Fig. 12: Gadget coverage comparison of ODDFUZZ, ODD-FUZZ-RM, ODDFUZZ-CG, and ODDFUZZ-DG. The x-axis is 34 known gadget chains listed in Table IV. The y-axis is the gadget coverage.

```
1  public class PriorityQueue<E> implements Serializable {
2    transient Object[] queue;
3    private final Comparator<? super E> comparator;
4    private void readObject() /*Source*/
5    private void heapify()  /*2nd Gadget*/
6    private void siftDown() /*3rd Gadget*/
7    private void siftDownUsingComparator() /*4th Gadget/

8  /*com.tangosol.util.comparator*/
9  public class ExtractorComparator<T> {
10   private ValueExtractor<? super T, ? extends Comparable> m_extractor;
11   public int compare(T o1, T o2) { /*5th Gadget/
12     Comparable a1 = o1 instanceof Entry ?
          (Comparable)((Entry)o1).extract(this.m_extractor) :
             (Comparable)this.m_extractor.extract(o1);
13   ...}}

14 /*com.tangosol.util.extractor*/
15 public class ChainedExtractor<T, E> {
16   public E extract(Object oTarget) { ...  /*6th Gadget*/
17     for(int c = aExtractor.length; i < c && oTarget != null; ++i) {
18       oTarget = aExtractor[i].extract(oTarget);
19   ...}

20 /*oracle.eclipselink.coherence.integrated.internal*/
21 public class FilterExtractor {
22   protected AttributeAccessor attributeAccessor;
23   public Object extract(Object obj) { ... /*7th Gadget*/
24     attributeAccessor.getAttributeValueFromObject(obj);
25   ...}}

26 /*org.eclipse.persistence.internal.descriptors*/
27 public class MethodAttributeAccessor {
28   protected transient Method getMethod;
29   public Object getAttributeValueFromObject(Object anObject) {/*8th Gadget*/
30     return this.getAttributeValueFromObject(anObject, (Object[])null); }
31   protected Object getAttributeValueFromObject(Object anObject,
                                Object[] parameters) { /*9th Gadget*/
32     return this.getMethod.invoke(anObject, parameters); /*Sink*/
33   ...}}
```

(a) A simplified code snippet of the gadget chain for CVE-2020-14756 in WebLogic.

(b) An exploitable gadget chain for 13a.

Fig. 13: A previously unknown vulnerability found by ODD-FUZZ.

TABLE IV: Evaluation results of ODDFᴜᴢᴢ on each gadget chain from ysoserial. **# Gadgets** denotes the number of gadgets in each gadget chain. `Identified` and `Validated` respectively represents whether the gadget chain can be statically identified and dynamically validated.

ID	Gadget Chain	Affected Version	# Gadgets	ODDFᴜᴢᴢ Identified	ODDFᴜᴢᴢ Validated	GadgetInspector	SerHybrid
1	*AspectJWeaver*	aspectjweaver-1.9.2	9	✓	✗	-	-
2	*BeanShell1*	bsh-2.0b5	6	-	-	-	-
3	*C3P0*	c3p0-0.9.5.2	6	✓	✓	-	-
4	*Click1*	click-nodeps-2.3.0	10	✓	✓	-	-
5	*Clojure*	clojure-1.8.0	10	✓	✓	✓	-
6	*CommonsBeanutils1*	commons-beanutils-1.9.2	5	✓	✓	-	-
7	*CommonsCollections1*	commons-collections-3.1	7	✓	✗	✓	-
8	*CommonsCollections2*	commons-collections4-4.0	13	✓	✓	-	✓
9	*CommonsCollections3*	commons-collections-3.1	13	✓	✗	-	-
10	*CommonsCollections4*	commons-collections4-4.0	15	✓	✓	-	-
11	*CommonsCollections5*	commons-collections-3.1	8	✓	✓	-	-
12	*CommonsCollections6*	commons-collections-3.1	10	✓	✓	-	✓
13	*CommonsCollections7*	commons-collections-3.1	9	✓	✓	-	-
14	*FileUpload1*	commons-fileupload-1.3.1	3	-	-	-	-
15	*Groovy1*	groovy-2.3.9	10	-	-	-	-
16	*Hibernate1*	hibernate-core-4.3.11.Final	7	✓	✓	-	-
17	*Hibernate2*	hibernate-core-4.3.11.Final	9	✓	✓	-	-
18	*JBossInterceptors1*	jboss-interceptor-core:2.0.0.Final	5	-	-	-	-
19	*JRMPClient*	JDK-1.7	13	-	-	-	-
20	*JRMPListener*	JDK-1.7	9	-	-	-	-
21	*JSON1*	json-lib:jar-jdk15:2.4	22	-	-	-	-
22	*JavassistWeld1*	javassist-3.12.1.GA	5	-	-	-	-
23	*Jdk7u21*	JDK-1.7	11	-	-	-	-
24	*Jython1*	jython-standalone-2.5.2	5	✓	✗	✓	-
25	*MozillaRhino1*	js-1.7R2	8	✓	✓	-	-
26	*MozillaRhino2*	js-1.7R2	12	✓	✓	-	-
27	*Myfaces1*	myfaces-impl-2.2.9	4	-	-	-	-
28	*Myfaces2*	myfaces-impl-2.2.9	6	-	-	-	-
29	*ROME*	rome-1.0	15	✓	✓	-	-
30	*Spring1*	spring-core:4.1.4.RELEASE	11	-	-	-	-
31	*Spring2*	spring-core:4.1.4.RELEASE	12	-	-	-	-
32	*URLDNS*	JDK	7	✓	✓	-	-
33	*Vaadin1*	vaadin-server-7.7.14	10	✓	✓	-	-
34	*Wicket1*	wicket-util-6.23.0	3	-	-	-	-

TABLE V: List of previously unknown deserialization vulnerabilities discovered by ODDFᴜᴢᴢ.

No.	Application	Version	Impact	Status	CVE-ID
1	WebLogic	12.2.1.4.0	RCE	Patched	CVE-2020-14756
2	WebLogic	12.2.1.4.0	RCE	Patched	CVE-2020-14825
3	WebLogic	12.2.1.4.0	RCE	Patched	CVE-2021-2135
4	Sonatype Nexus	3.25.0	RCE	Patched	CVE-2020-15871
5	Apache Dubbo	2.7.7	RCE	Patched	CVE-2020-11995
6	ProtoStuff	1.8.0	RCE	Reported	