Efficient Detection of Java Deserialization Gadget Chains via Bottom-up Gadget Search and Dataflow-aided Payload Construction

Bofei Chen[1], Lei Zhang[1*], Xinyou Huang[1], Yinzhi Cao[2], Keke Lian[1*], Yuan Zhang[1*], Min Yang[1*]
1: Fudan University, {bfchen22, xinyouhuang22}@m.fudan.edu.cn
1: Fudan University, {zxl, kklian20, yuanxzhang, m_yang}@fudan.edu.cn*
2: Johns Hopkins University, yinzhi.cao@jhu.edu

***Abstract*—Java Object Injection (JOI) is a severe type of vulnerability affecting Java deserialization, which allows adversaries to inject a well-crafted, serialized object, thus triggering a series of chained internal methods (called gadgets) and then achieving attack consequences such as Remote Code Execution (RCE). Prior works studied the problem of detecting and chaining gadgets for JOI vulnerability using static search for possible gadget chains and dynamic construction of payload via fuzzing. However, prior works face two following challenges: (i) path explosion in static gadget search and (ii) a lack of fine-grained object relations connected via object fields in dynamic payload construction.**

In this paper, we design and implement a novel Java deserialization gadget detection framework, called JDD. On one hand, JDD solves the static path explosion problem by a bottom-up approach, which first looks for gadget fragments and then chains gadget fragments from sinks to sources. The approach reduces maximum static search time from exponential to polynomial, i.e., from $O(eM^n)$ to $O(M^2n^3 + enM)$, where n is the number of dynamic function calls in a gadget chain, M is the average number of dynamic function call candidates, and e is the number of entry points. On the other hand, JDD constructs a so-called Injection Object Construction Diagram (IOCD), which models the dataflow dependencies between injection objects' fields to facilitate dynamic fuzzing. Our evaluation of JDD upon six real-world Java applications reveals 127 zero-day, exploitable gadget chains with six Common Vulnerabilities and Exposures (CVE) identifiers assigned. We also responsibly reported these vulnerabilities to application developers and obtained their acknowledgments and confirmations.

1. Introduction

Java serialization and deserialization greatly facilitate the cooperation and collaboration of different Java systems (e.g., server and client), allowing different Java programs to conveniently exchange and share data and code. Despite their easiness and powerfulness, one well-known and serious security vulnerability faced in Java deserialization is an implicit attack surface called Java Object Injection (JOI). The JOI vulnerability enables a remote attacker to inject a well-crafted serialized object, which triggers a chain of internal Java methods (called gadgets), finally achieving a wide range of severe attack consequences, e.g., remote code execution (RCE) and denial of service (DoS) attacks.

In past years, many works [1]–[4] have focused on the problem of JOI, and proposed several JOI vulnerability detection techniques. For example, ODDFuzz [2] statically searches for possible gadgets via a Depth First Search (DFS) strategy, and then dynamically fuzzes the target program as a greybox for verifying gadget chains. However, existing work still fell short in two major limitations, impacting their practicality and effectiveness.

First, prior work often adopted a top-down static approach for checking gadgets extensively from a source to a sink for potential paths. However, such a search often suffers from path explosion, especially when common Java methods such as `equals` and `put` are involved in the gadget chain. For example, our experiment shows that the static analysis approach like ODDFuzz cannot finish analyzing a small 3.6MB Java application. Fundamentally, the search complexity is exponential, i.e., $O(eM^n)$, where n is the number of dynamic function calls in a gadget chain, M is the average number of dynamic function call candidates, and e is the number of entry points.

Second, many of prior tools conducted static analysis against JOI vulnerability detection, thus inevitably having high false positives. Beyond these, ODDFuzz additionally introduced dynamic fuzzing for reducing false positives. However, ODDFuzz did not consider the constraints that should be satisfied in each gadget. In particular, the fine-grained data flows between different object fields in each gadget, leading to the imprecision of object structure and thus an incorrect payload.

In this paper, we design a novel gadget chain detection framework, called JDD (Java Deserialization Vulnerability Detector), to statically detect possible gadget chains using a bottom-up approach and dynamically generate payloads, i.e., exploitable injection objects, relying on dataflow relations between object fields. JDD addresses the aforementioned two challenges as follows.

First, JDD addresses the path explosion challenge via a bottom-up search strategy. JDD first searches for possible gadget fragments and then chains fragments together from sinks to sources. Our key observation here is that a top-down search repeats the analysis of the same low-level gadget

fragment when a top-level gadget changes, leading to a huge redundancy and a waste of analysis time. Instead, the bottom-up search adopted by JDD finds gadget fragments that can be reused by different top-level gadgets in the chain. Therefore, the search complexity changes from exponential to polynomial, i.e., from $O(eM^n)$ to $O(M^2n^3 + enM)$. More importantly, JDD is able to reuse existing gadget fragments discovered in previous vulnerabilities, which further speeds up the search process.

Second, JDD addresses the challenge of the lack of object field relations via a novel data structure, called Injection Object Construction Diagram (IOCD), to better represent the dataflow dependencies between injection objects' fields. The high-level idea is that JDD follows the call sequence in a statically-discovered gadget chain to construct dataflow dependencies between possible injection objects' fields as an IOCD. Such an IOCD is further used by JDD in dynamic fuzzing to exploit the JOI vulnerability.

We evaluate JDD upon six popular Java applications in their latest version, which finds 127 zero-day exploitable gadget chains with six Common Vulnerabilities and Exposures (CVE) identifiers assigned. We also responsibly reported our findings of all 127 gadget chains to related developers and received their confirmation. For example, Dubbo's developers not only acknowledged the gadget chains but also admitted that it is almost "impossible" for them to recognize all possible gadget chains, which emphasizes the urgency of our automated tool.

We also compare JDD with the state-of-the-art approach, namely ODDFuzz, in a well-known benchmark (i.e., the ysoserial repository [5]) with 34 confirmed gadget chains. JDD missed only seven of 34 confirmed with a False Negative Rate (FNR) of 20.6% and identified 91 previously-unknown gadget chains (which were confirmed as correct in manual analysis). As a comparison, OddFuzz only detected 16 gadget chains, resulting in an FNR of 52.9%, and it did not discover any previously-unknown ones.

We summarize the contributions of this paper as below:

- JDD solves the path explosion problem of static search for chained gadgets via a bottom-up approach that discovers intermediate gadget fragments and then chains them together from sinks to sources.
- JDD constructs the so-called Injection Object Construction Diagram (IOCD) to better model dataflow dependencies between object fields and assist dynamic fuzzers for payload construction.
- JDD discovered 127 zero-day gadget chains of real-world Java applications in their latest version and outperforms ODDFuzz in detecting legacy gadget chains.

2. Overview

In this section, we first provide a brief background on JOI attacks in Section 2.1, then describe Java deserialization gadget chains in Section 2.2 using a real-world motivating example, and then present research challenges faced by state-of-the-art approaches in detecting this example in Section 2.3.

2.1. Background of JOI Attacks

Java Naming and Directory Interface (JNDI) injection is a commonly used technique in JOI attacks. The attack occurs because the JNDI interface's `lookup` method can remotely load a malicious Java class based on its parameters (e.g., an attacker-controlled URL), thereby executing arbitrary code in the victim server. A typical scenario for this attack involves a web service running on the victim server, listening on a port and deserializing data received on the port. During deserialization, if there is a JOI vulnerability, the program execution will invoke some security-sensitive methods based on the injected object.

In addition to JNDI injection attacks, common methods of exploiting JOI vulnerabilities to attack remote servers include dynamic code loading (such as using `URLClassLoader.loadClass` to load remote class files or using `ClassLoader.defineClass` to directly load bytecode files), and command execution (such as `Runtime.exec`), etc. They all require: (1) An execution path that can invoke a security-sensitive method that can execute malicious code (or upload arbitrary files, deny service, etc.) during the deserialization process, called *gadget chain*; (2) A serialized object that drives the execution of the gadget chain, called *injection object*.

Thus, to detect JOI vulnerabilities, JDD has two major detection goals: (1) Uses static analysis to find potential gadget chains in the victim server; (2) Dynamically generates exploitable injection objects to verify the exploitability of gadget chains. As a result, the exploitable gadget chains detected by JDD can be used to help developers specify a more complete blacklist defense mechanism to better defend against JOI attacks.

2.2. A Motivating Example

Figure 1 illustrates a zero-day gadget chain found by JDD in `sofa-rpc` [6], i.e., popular Java libraries that are widely used in the server-side cloud of many enterprise companies, e.g., Alipay. The gadget chain starts from a Java deserialization method (e.g., `HashMap.put()` in Line 4, shown in Figure 1). The chain ends up with a Java reflection call at Line 54 and then launches a JNDI injection attack at Line 59. Note that each Java method call along the chain is considered as a *gadget*, e.g., the `put()` method at Line 4. Then, Java method calls between two dynamically-dispatched methods are considered as a *gadget fragment*, e.g., the combination of methods `get()` and `getFromHashtable()` in Figure 1 as Fragment IV, as the latter is statically determined. We also illustrate the exploit code in Figure 2. An adversary serializes the returned object in Line 22 of Figure 2 and sends it to the server, which triggers the gadget chain in Figure 1 for exploitation.

We first describe the gadget chain with five different gadget fragments of Figure 1 in detail. The gadget starts from Fragment I (Line 1), where the `put()` method (Line 4) of an object `HashMap` is invoked upon deserialization. If two elements in the `HashMap` have the same hash value,

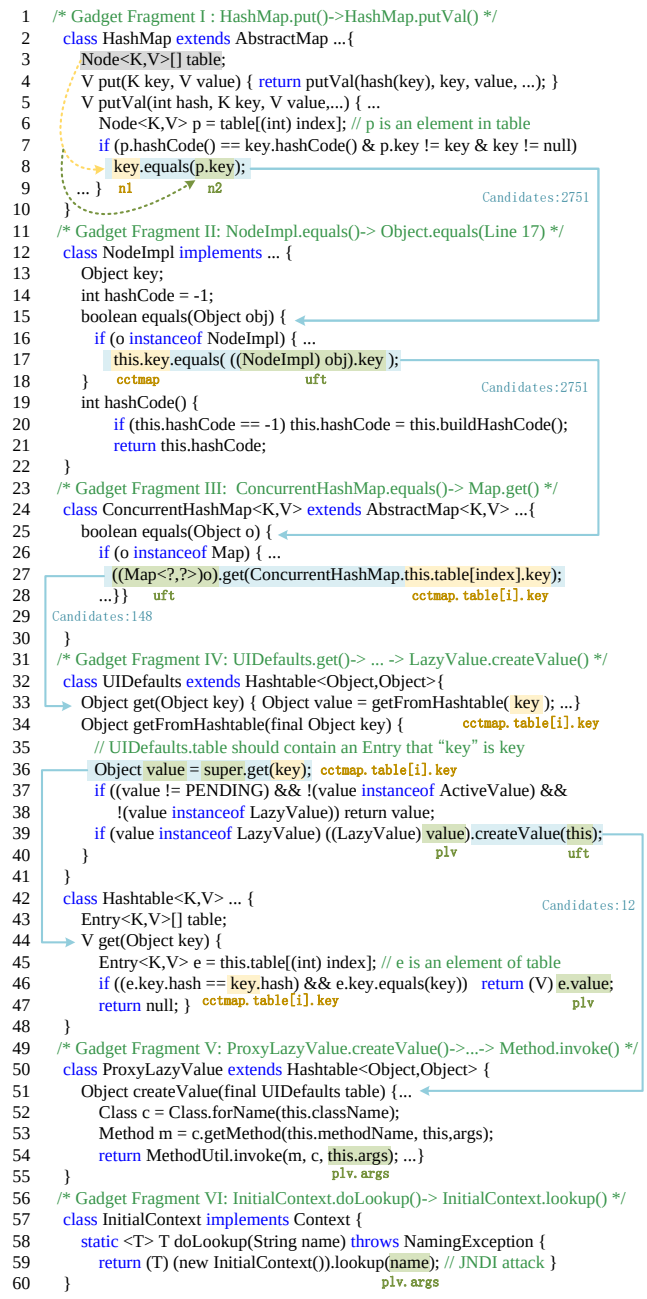

Figure 1: A Motivating Example of a Zero-day Gadget Chain with Five Fragments (Note that code is simplified for easy understanding.)

the `equals()` method (Line 8) will be invoked, which may have a polymorphic implementation in `NodeImpl` class, thus triggering Fragment II (Line 10). Then, the field `key` of the `NodeImpl` class triggers another `equals()` method (Line 17), which may have a polymorphic implementation in the `ConcurrentHashMap` class, thus triggering Fragment III (Line 25). Next, the `get()` method has a polymorphic implementation in the `UIDefaults` class, leading to Fragment IV, which further calls the `getFromHashtable()` method (Line 34). Lastly, the `createValue()` method calls at Line 39 invokes the polymorphic method definition in Line 51 belong to Fragment V, which calls Java reflection at Line 54 and thus the JNDI attack at Line 59 of Fragment VI.

```
1  // Gadget Fragment IV
2    UIDefaults uft = new UIDefaults();
3  // Gadget Fragment V
4    Object plv = createWithObjectNoArgsConstructor(
5            Class.forName("javax.swing.UIDefaults$ProxyLazyValue"));
6    uft.put("aaa", plv)
7  // Gadget Fragment VI
8    setFieldValue(plv, "className", "security-sensitive class name");
9    setFieldValue(plv, "args", new Object[]{"malicious URL"});
10   setFieldValue(plv, "methodName", "security-sensitive method name");
11 // Gadget Fragment III
12   ConcurrentHashMap cctmap = new ConcurrentHashMap();
13   cctmap.put("aaa", "any");
14   // Gadget Fragment II
15   NodeImpl n1 = createWithObjectNoArgsConstructor(NodeImpl.class);
16   NodeImpl n2 = createWithObjectNoArgsConstructor(NodeImpl.class);
17  setFieldValue(n1, "hashCode", 3);
18  setFieldValue(n2, "hashCode", 3);
19   setFieldValue(n1, "key", cctmap);
20   setFieldValue(n2, "key", uft);
21   // Gadget Fragment I
22   HashMap map = Gadgets.makeMap(n2, n1); // put n1 and n2 into a HashMap
```

Figure 2: Exploit Code of Our Motivating Example in Figure 1 (An adversary serializes the returned object in Line 20 and sends it to the server).

Next, we describe how an adversary utilizes the gadget chain with the exploit code in Figure 2. An adversary first instantiates a `HashMap` object at Line 22 of Figure 2 to trigger the `put()` method of Fragment I. Then, two `NodeImpl` objects are instantiated with the same `hashCode` (Lines 15–18 of Figure 2) so that the `equals()` method in Fragment II is invoked. Next, the adversary assigns `n1.key` as a `ConcurrentHashMap` object (Lines 12–13 and 19 of Figure 2) to trigger another `equals()` method in Fragment III. After that, the adversary assigns `n2.key` as a `UIDefaults` object (Lines 2 and 20 of Figure 2) to trigger the `get()` method in Fragment IV. Lastly, the adversary manipulates the elements stored in `uft.table` to return a `ProxyLazyValue` object (Line 6 of Figure 2), which implements `LazyValue` interface with three values to trigger Fragment V. Specifically, the adversary assigns the method `InitialContext.doLookup()` as the `className` and the `methodName` together with the correct `args` (Lines 8–10 of Figure 2) to trigger Fragment VI.

2.3. Challenges and Solution Overview

We now describe two challenges faced by prior works in their static and dynamic analysis using the motivating example presented in Figure 1. Then, we present the solutions proposed by us in designing `JDD` to solve these two challenges.

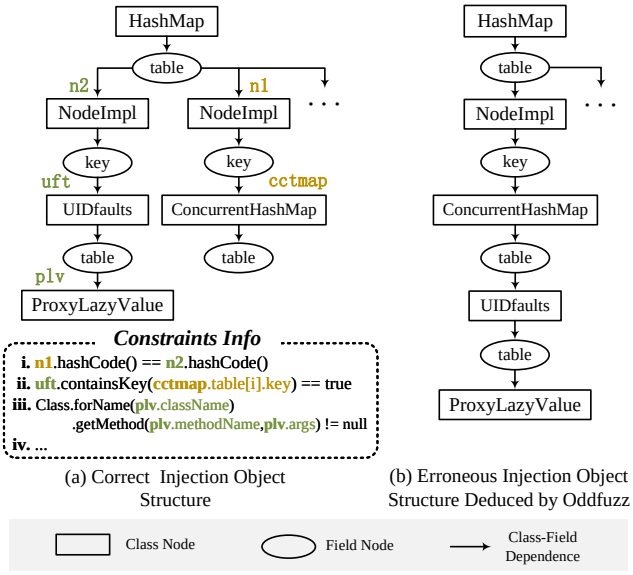

Figure 3: Injection Object Structure of Motivation Example.

Challenge I: Static Path Explosion. The first challenge is that the number of all possible paths between sources and sinks is exponential, leading to path explosion. Let us use Figure 1 as an example to describe the challenge. The number of potential candidates for the `equals()` method between Fragments I and II could be 2,751 and the same applies to the other `equals()` method between Fragments II and III. Then, the candidate number is 148 for `get()` between Fragments III and IV, and 12 for `createValue()` between Fragments IV and V. Therefore, if a Depth-first-search (DFS) is used to find a gadget chain like what ODDFuzz does, the total number execution path could be 2751×2751×148×12 = 13,440,769,776, which is impossible to search from.

Our Solution: Bottom-up Gadget Search. Our key observation is that a top-down search like a DFS will analyze some methods multiple times, leading to redundancies. For example, one `get()` method candidate between Fragments III and IV may be analyzed once for one `equals()` method candidate between Fragments II and III and then again for another candidate between Fragments II and III. Instead, `JDD` analyzes all possible gadgets between sources and sinks once and forms them into gadget fragments. Then, `JDD` adopts a bottom-up search from the sink to the source, which chains all the fragments into a chain. Specifically, `JDD` only needs to analyze (12 + 2751 + 148) = 2,911 program execution paths and at most (2911 × 2911 - 2751 × 2751 - 148 ×148 - 12 × 12) × 2 + 2751 = 1,770,495 times fragment linking check for our motivating example in Figure 1, which is only 0.01% of the top-down approach, significantly reducing the static gadget search space.

Challenge II: Parallel and Embedded Injection Object Structure. The second challenge is that the payload, i.e., the injection object, may have complex structures, such as embedded or parallel objects. Figure 3 (a) shows the injection object structure for our motivating example in Figure 1. `UIDfaults` and `ConcurrentHashMap` are two `keys` under `NodeImpl` objects in a correct payload. However, prior works, such as ODDFuzz, only consider the class hierarchy inferred from the gadget chain and therefore they will generate a wrong object structure as shown in Figure 3 (b) because the `get()` method of `UIDfaults` is invoked after `ConcurrentHashMap`.

Our Solution: Dataflow-aided Construction of Injection Object Construction Diagram (IOCD). Our key observation is that different injection objects, e.g., their fields, are connected via dataflows. Specifically, two dotted lines in different colors of Figure 1 show such two dataflows. `key` (Line 8) flows to `this.key` (Line 17) and then `this.table[index].key` (Line 27); then, `p.key` (Line 8) flows to `((NodeImpl) obj).key` (Line 17) and then `(Map<?,?>)o` (Line 27). Therefore, `JDD` infers that there are two `NodeImpl` objects and `UIDfaults` is under the `key` of one `NodeImpl` object instead of the `table` of `ConcurrentHashMap`. More specifically, `JDD` constructs an IOCD like Figure 3 (a) following such dataflows, which can be used for follow-up fuzzing.

3. Design

In this section, we describe the system architecture of `JDD` and then the detailed steps.

3.1. System Architecture

We show an overview of `JDD` 's architecture with two main stages in Figure 4. In Stage I, `JDD` detects possible gadget chains and then in Stage II, `JDD` generates injection objects to exploit the detected gadget chains for validation. Specifically, there are five steps. In Step 1, `JDD` identifies deserialization entry points. Then, in Step 2, `JDD` starts from entry points to identify gadget fragments with static analysis. After that, `JDD` links gadget fragments to construct gadget chains using a bottom-up approach, which finishes Stage 1 with possible gadget chains. Next, in Step 4, `JDD` constructs Injection Object Construction Diagram (IOCD) based on the injection object related constraints. Lasty, in Step 5, `JDD` utilizes IOCD-enhanced directional Fuzzing to verify gadget chains' exploitability.

We now describe how each step works for our motivating example. Here is Stage I. First, in Step 1, `JDD` starts with identifying the entry points in `sofa-rpc` [6], and then use them as the sources to begin our static taint analysis. Second, in Step 2, `JDD` traverses the control- and data-flow graph of the target program starting from each source (i.e., entry points in the beginning) until a dynamic method invocation and considers this as a fragment. Then, `JDD` treats the implementation of the dynamic method as the new source until the next dynamic method invocation and considers code in between as a new fragment. During this process, `JDD` identifies some gadget fragments containing a sink, e.g., `MethodUtil.invoke()` in Gadget

Fragment V. Then, in Step 3, JDD uses this fragment to find the gadget fragment IV that has a dynamic method invocation `LazyValue.createValue()` and the object `value` that is used to invoke the `createValue()` method based on previous taint analysis result. Next, JDD uses the Gadget Fragment IV as the new start to find the Gadget Fragment III and repeats this procedure until it reaches a source point, resulting in a gadget chain. After chaining the gadget fragments I-V, to complete the gadget chain, JDD will independently gather potential fragments that could execute remote attacker-controlled code, for example, the gadget fragments VI which contains a JNDI capability. Since the gadget fragment V has a reflection capability and the parameter is controlled by the attacker, JDD would further connect it to the gadget fragment VI.

We then describe Stage II. In Step 4, JDD extracts the constraints for gadget chain chaining, which include (1) structured constraints (i.e. the class hierarchy relationships between object and field instances), (2) field dependency constraints (e.g., the hash code of *n1.key* and *n2.key* should be the same for our example), and (3) conditional branches. Note that JDD labels constraints shared by different execution paths as *dominator*, meaning that they are required by all the execution paths. After extracting these constraints, JDD uses a novel data structure, termed Injection Object Construction Diagram (*IOCD*) to model dataflow dependencies between object fields and guide dynamic fuzzing. Lastly, in Step 5, JDD instruments the collected constraints of the target program and uses the number of covered dominator constraints to guide the fuzzing for generating an injection object that can exploit this gadget chain. Specifically, JDD first initializes an instance object based on the class hierarchy in the IOCD graph. Then, for assigning adequate value for each field in the object, JDD utilizes a constraint solver to solve the dominator constraints. Then, JDD uses this object as the seed to drive the directional fuzzing and uses the number of covered dominator constraints to evaluate its energy.

3.2. Step 1: Identifying Deserialization Entry Points

The first step of our analysis is to identify the entry points of deserialization in a given Java program. JDD extract such entry points from both the deserialization methods of Java language and popular Java deserialization protocols as shown in Table 1. There are two categories of entry methods: (i) deserialization methods provided by Java language, e.g., `readObjectNoData()`, `readExternal()` and `readObject()`, and (ii) interfaces provided by popular Java deserialization protocols, e.g., `Map.put()` for Hessian protocol. Note that since many of these entry points are defined as interfaces, JDD further identifies and analyzes their overridden or implementation methods as the entry points.

Table 1: Popular Deserialization Protocols

Protocol	Entry Points	Supported Dynamic Feature	Unserializable Class Support
JDK	readObject() readObjectNoData() readResolve() readExternal()	Polymorphism Reflection Proxy	NO
T3/IIOP	readObject() readObjectNoData() readResolve() readExternal()	Polymorphism Reflection Proxy	NO
Hessian	Map.put() toString()	Polymorphism Reflection	YES
Hessian-lite [7]	Map.put()	Polymorphism Reflection	NO
Hessian-sofa [8]	Map.put() toString()	Polymorphism Reflection	YES
XStream	readObject() Map.put()	Polymorphism Reflection Proxy	YES

3.3. Step 2: Identifying Gadget Fragments with Static Taint Analysis

In this step, JDD detects useful gadget fragments that could be chained together to construct gadget chains from entry points using static analysis. There are two substeps: (i) gadget fragment formation and (ii) data-flow analysis within a fragment. First, JDD starts from each source, i.e., entry points or implementations of previous dynamic method invocation, until the next dynamic method invocation and treats code in between as a fragment. Second, JDD also performs a data-flow analysis to record taint propagation from the parameters of the first method in the fragment. Specifically, we design a *fragment-based summary* that, for each gadget fragment, records the pollution behavior between its first method and the last method parameters and maintains a sink-reachable condition record. Our purpose of converting inter-procedure analysis of methods into jumps between gadget fragments is to flexibly simulate dynamic method invocations and reduce the computational complexity of inter-procedure searches.

Next, we first describe the definitions of gadget fragments and then a classification of gadget fragments.

Gadget Fragment Definition. A gadget fragment is a straight-line gadget sequence (i.e., methods could be executed in Java deserialization), and its execution starts from the fragment's head to its endpoint. For each fragment, its endpoint commonly is a security-sensitive method or a dynamic method invocation that points to another fragment, for example, `Object.equals()` could connect to almost all the method "equals()" implemented by any class. During deserialization, this kind of dynamic method invocation indeed introduces many possibilities of the program's execution direction, thus they work like the jump/goto instructions.

In detail, the methods in a gadget fragment have:

- *Head*, which is the entry method of this fragment and there exist some dynamic method invocations that could jump to this method.
- *End*, which is the exit method of this fragment and commonly a dynamic method invocation or security-sensitive method.

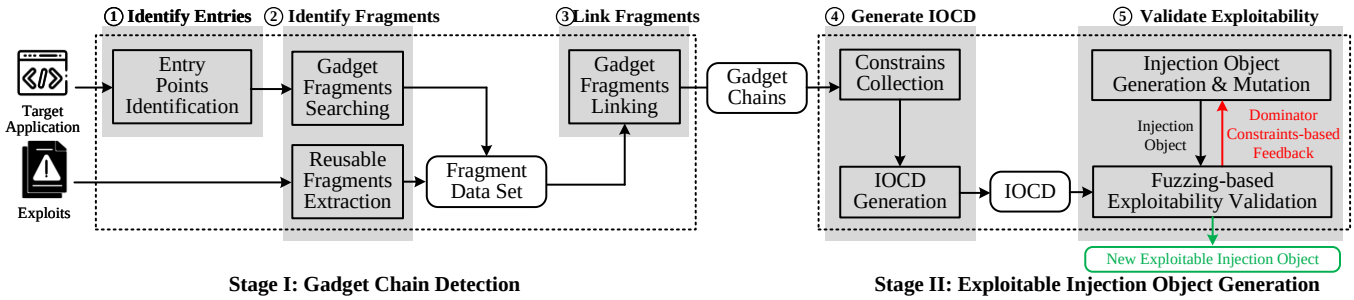

Figure 4: The Overall Architecture of JDD with Two Stages.

- Other gadgets, which are non-dynamic methods and executed in sequence during deserialization to connect the head and end.

Dynamic Method Invocation Types for Gadget Fragments. There are the following three types of dynamic method invocations in gadget fragments:

- *Polymorphic*: JDD recognizes the inheritance hierarchy of the target program's classes and identifies the overridden methods. Then, when a method in a parent class is invoked, JDD connects it to the corresponding overridden methods implemented by its child classes. For example, considering a class `B` that overrides method "m" of class `C`, any fragment, where *End* is `C.m()`, could jump to other fragments, where *Head* is `B.m()`.
- *Dynamic Proxy* [9]: Java language supports objects under type `interface`, `Object` and other generic types to be implemented as a dynamic proxy instance, and when the instance invokes any method, it would be redirected to a specific invocation handler, i.e., the `invoke()` method implemented in this proxy. Such handlers commonly deploy method routes based on the properties (e.g. method name) of the trigger method. Thus, for each handler, JDD conducts a path-sensitive analysis to understand its method routs for ensuring the trigger method will not connect to a wrong execution path in invocation handlers. In practice, JDD first identifies gadget fragments in all execution paths inside an invocation handler and then labels the gadget fragments with corresponding method property requirements based on their execution paths.
- *Reflection* [10]: Unlike other dynamic features, Java reflection could invoke almost all methods implemented in the program based on its parameters. JDD first checks if the reflection's parameters are attacker-controlled (so that JOI can lead to RCE) and then limits it to connect fragments that contain code execution capabilities. Note that JDD allows reflection to connect any gadget in its successor fragments but prefers two kinds of gadgets, based on the analysis of existing gadget chains: (1) methods with no parameters, especially *getter* or *is* methods, and implementation of interface methods, and (2) methods with one parameter of `Boolean` or `String` type.

3.4. Step 3: Linking Gadget Fragments to Construct Gadget Chains using a Bottom-up Approach

Upon completing the search of gadget fragments, JDD performs a bottom-up search to chain fragments together, thereby constructing all potential gadget chains as shown in Algorithm 1. First, JDD obtains three types of fragments: (1) *Source Fragments*, whose head is a source, e.g., the Gadget Fragment I in Figure 1; (2) *Free-State Fragments*, which records method exection sequence between two dynamic method invocations, e.g., the Gadget Fragments II-IV in Figure 1; (3) *Sink Fragments*, whose end is a sink, e.g., Fragments V, VI in Figure 1. Then, JDD constructs potential gadget chains by chaining *Sink Fragments* with *Free-State Fragments* and then *Source Fragments* with the following steps. Note that JDD needs to consider taint requirements and method invocation conditions corresponding to different dynamic features. For example, sink fragments commonly require specific parameters should be controlled by attackers. Then, when finding precursors for Sink Fragments, JDD would iteratively check if the corresponding parameters could be tainted.

We now describe the details in Algorithm 1. First, JDD connects the *Free-State Fragments* that can transition to *Sink Fragments*. As shown in Algorithm 1, during each round (Line 2–19), JDD traverses each fragment $frag_{fs}$ in *Free-State Fragments*. Based on the summary of $frag_{fs}$, JDD checks whether the linking conditions to sink fragments are satisfied. If so, $frag_{fs}$ is added to *newSinkFragments* as the *Sink Fragment* (Line 8–9) for the next iteration, while keeping track of the taint requirements to be satisfied for the connection (merging them if they are identical to the already recorded taint requirements) (Line 11). Since each iteration can pick out all fragments that can potentially connect to the current sink fragments, the iteration process terminates if there are no new Sink Fragments, i.e., when *newSinkFragments* is empty (Line 15–16) or reaching the maximum number of search attempts. Next, the *Source Fragments* are traversed, and the same approach is employed to detect connectable successor fragments, thereby obtaining all potential gadget chains (Line 20–26).

Lastly, assuming there are n' distinct dynamic method invocations, let us explain why the upper limit for the maximum number of iterations is n'. If the number of

Algorithm 1 Gadget Fragments Linking

Input: Sink Fragments Dataset D_{sk}
Input: Source Fragments Dataset D_{src}
Input: Free-state Fragments Dataset D_{fs}
Output: Gadget Chain DataSet S

```
// n' is the number of distinct dynamic method invocations in context.
for i ← 1 to n' do
    newSinkFragments ← ∅
    for frag_fs in D_fs do
        for frag_succ in D_sk.getFragStartWith(frag_fs.end) do
            // Checking whether the taint requirements are satisfied
            if isLinkable(frag_fs, frag_succ) then
                newSinkFragments ← newSinkFragments ∪ frag_fs
                D_sk.add(frag_fs)
                // link frag_fs with frag_succ and merge taint requirements
                updateTaintRequirements(frag_fs, frag_succ)
            end if
        end for
    end for
    if newSinkFragments == ∅ then
        break
    end if
    D_sk ← newSinkFragments
end for
for frag_src in D_src do
    for frag_succ in D_sk.getFragStartWith(frag_src.end) do
        if isLinkable(frag_src, frag_succ) then
            S ← S ∪ link(frag_src, frag_succ)
        end if
    end for
end for
return S
```

iterations exceeds n', the longest fragments will necessarily contain candidates with two fragments whose ends are the same dynamic method. Since the attacker has the ability to invoke any candidate with the same dynamic method, the attacker can directly supply the class implementation in the latter call to the first to reduce the total length, so the path between two identical dynamic method invocations is redundant.

Time Complexity. JDD reduces the static search time from $O(eM^n)$ (i.e., those of state of the art, e.g., ODDFuzz [2]) to $O(M^2n^3 + enM)$ according to Theorems 1 and 2 below.

Theorem 1. *The search complexity of Algorithm 1 is $O(n^3M^2+enM)$, where n is the number of dynamic method invocations, M is the average number of candidates for these invocations, and e is the number of entry points.*

Proof. See Appendix A.1. □

Theorem 2. *The search complexity of ODDFuzz [2] is $O(eM^n)$, where all notations follow Theorem 1.*

Proof. See Appendix A.2. □

3.5. Step 4: Constructing IOCD based on Injection Object related Constraints

There are two substeps here: (i) constraint extraction and (ii) IOCD generation. First, JDD follows the call sequence of this gadget chain to extract the execution paths, and then uncovers three types of constraints that affect inputs:

- Class hierarchy relationships between object and field instances. After JDD performs bottom-up fragment linking, it will perform top-down class hierarchy inference according to the fragment connection sequence. Specifically, for each gadget fragment, JDD first uses dataflow analysis to determine which of its fields is linked to the next fragment, and then uses the header method of the subsequent fragment to determine the actual type of the field. For example, in the Gadget Fragment I in Figure 1, JDD finds the `HashMap` object has a field `table` (Line 3) and `table` should store two objects (i.e. *n1*, *n2*) based on the dataflow to Line 8. Another important problem here is to identify the exact class type of these fields, considering many of them are defined as generic type or `Object`, e.g. the *key* in Line 8, the field `key` of *n1* and *n2* (i.e. *this.key*, *obj.key*) in Line 17, etc. To solve this problem, JDD continues the dataflow tracing and finds that the *key* connects Fragment I with II, and the header method of Fragment II is `NodeImpl.equals`. Thus, *key* should be a `NodeImpl` instance. Similarly, *this.key* and *obj.key* connect Fragment II with III and Fragment III with IV, respectively. Therefore, JDD infers that the field `key` of two `NodeImpl` instances, *n1*, *n2* are `ConcurrentHashMap` and `UIDefaults` instances, as indicated by the header methods of successor Fragment III and IV. Based on the approach, JDD ultimately deduces the correct class hierarchy of the injection object as illustrated in Figure 3 (a).
- Conditional branches related to fields, which exist in the execution paths of the gadget chain, and note that JDD labels constraints shared by different execution paths as *dominator* meaning that they are required by all the execution paths. During the process of dataflow tracing, JDD concurrently collects conditional branch constraints related to fields by identifying if the variables checked in the constraints could be tainted by fields. Moreover, to ensure the program execution would not trigger an exception, these exist several implicit constraints should be satisfied. For example, a field could not be *null* when it invokes some methods. Another example is forced type casting, which requires the field should be an instance of specific type.
- Field dependency constraints, which encompass conditional constraints among fields, i.e. the constraints i-iii as shown in Figure 3. Note that sink points commonly require specific fields be controlled by the attacker to inject payloads. JDD would label these fields to check if they can be tainted by input. To find field dependency constraints, in general, JDD filters those conditional constraints that affect multiple fields and then labels these fields with related constraints as field dependencies. Furthermore, JDD also considers specific requirements for Java reflection. For example, the three fields `className`, `methodName`, and `args` used in Java reflection should ensure the target class contains the target method.

Second, after extracting these constraints, JDD uses a novel data structure, termed Injection Object Construc-

tion Diagram (*IOCD*) to model the object structure and dependencies of fields. There are two sub-steps. (i) JDD treats the instantiated objects contained in a gadget chain as *ClassNodes*. Each *ClassNode* stores the following information: the class name and the related *FieldNodes*, which represent the fields that may be utilized during the deserialization. The *FieldNode*, on the other hand, stores the relevant constraint information for these fields and marks whether these constraints are statically determinable as *dominator*. (ii) Based on the structural constraints, the *ClassNodes* are interconnected through *FieldNodes* using directed edges, thereby indicating the hierarchical relationships between instantiated objects. For example, in the motivation example, JDD will construct *ClassNodes* representing the instances of `NodeImpl`, `UIDfaults`, and `ConcurrentHashMap`. Then, following the structural constraints, JDD establishes edges from the `UIDfaults` and `ConcurrentHashMap` *ClassNodes* to the same *FieldNode* (i.e. `key`) of the `NodeImpl` *ClassNode*, which implies the existence of at least two NodeImpl instances. Thus, JDD will include an additional `NodeImpl` *ClassNode* to signify the existence of two `NodeImpl` instances, where the field `key` of each instance is assigned to `UIDfaults` and `ConcurrentHashMap` instances, respectively, as illustrated in Figure 3 (a).

3.6. Step 5: IOCD-enhanced Directional Fuzzing

After modeling the structure of injection object and the constraints should be satisfied, JDD further utilizes IOCD to enhance our directional fuzzing. In general, given a gadget chain, the fuzzing goal is to generate an injection object that could reach and exploit its sink point. Specifically, JDD first generates initial seeds based on the IOCD and then uses them to drive the fuzzing. In the exploration phase of directional fuzzing, JDD utilizes the number of covered dominator constraints to evaluate the seed's energy and guide the fuzzer to cover more dominator constraints until reach the sink point. In the exploitation phase of directional fuzzing, JDD only mutates the fields related to the sink point. During each field's mutation, JDD adjusts other related fields to fix the cross-field dependencies based on IOCD.

IOCD based Seed Generation. Considering the cross-field dependencies and nested object structure, JDD adopts the IOCD-guided fuzzer to generate objects with the correct class hierarchy while taking into account the constraints among fields. Specifically, JDD generates parameterless instances based on *ClassNodes* from the *IOCD* and establishes the class hierarchy of these instances according to directed edges, as illustrated in Figure 3. Subsequently, starting from the root *ClassNode* node, a Breadth-First traversal is performed on each *FieldNode*. JDD will extract *dominator* constraints of these *FieldNodes*, and invoke the constraint solver to generate appropriate values, which are then assigned to the corresponding fields. In situations where bidirectional edges exist between *FieldNodes*, indicating the existence of constraint dependencies between two fields, JDD extracts *dominator* condition constraints from the relevant *FieldNodes*. These constraints are then solved using a constraint solver to find values that simultaneously satisfy these constraints.

Dependency-aware Seed Mutation. IOCD significantly enhances the efficiency of fuzzing from the following three aspects: (1) Selecting appropriate fields for mutation; (2) Reducing the uncertain mutation space through constraint information; (3) Considering fields dependencies and nested object structure. That is when mutating a field, the constraint relationships between this field and other related fields or upper-level instance objects containing this field are considered. Detailed strategies can be found in Table 2.

(i) Mutation based on condition constraints. Firstly, based on the feedback information obtained from seed execution, JDD matches suitable condition constraints from IOCD that are derived from the specified program execution flow of gadget chains. And these condition constraints are utilized to selectively guide the structural mutation of injection objects. In order to record the selected condition constraint strategies, JDD defines a unique identifier and assigns a 2-bit flag to indicate the mutate strategy for each condition constraint. The *first bit* is used to indicate whether the constraint is used (0: not selected, 1: selected), and the *second bit* signifies whether the constraint is satisfied (true or false). With this identifier, JDD prefers to select unused strategies to cover more kinds of combinations of condition constraints. The following explains how to generate condition constraint branch strategies based on feedback information:

(1) If JDD resolves the `ClassCastException` or `NullPointerException` exception information, relevant fields causing the exception can be matched from *IOCD* based on error location information (class name, code line number), and JDD will require these fields to satisfy specific constraints, i.e. $field$!= $null$, $field\ instanceof\ X.class$.
(2) Extract all constraints between the current reached and the next *dominator* condition constraints, and select a mutation strategy based on each constraint's identifier.
(3) If there are no *non-dominator* condition constraints between the current and the next *dominator* condition constraints, random mutation strategies are selected for the current *dominator* and the preceding conditional constraints, note that *dominator* condition constraints do not alter the *second bit* flag.
(4) If the program has reached the sinks but has not triggered the expected malicious instructions, JDD first checks whether there are multiple ways to construct malicious data (i.e., constructed from different fields). If such variations exist, JDD uses the information recorded in *IOCD* to call different fields to construct malicious data. Otherwise, randomly changing the condition constraints strategies.
(5) If all combinations of strategies for condition constraints have been used, a 50% probability is assigned

Table 2: Mutation Strategy.

Property(*prop*) Type	**Constraint(*cnst*) Type**	**Example**	**Mutation Strategy**
Class	Class	if(*prop*==*cnst*)	set *prop* to *cnst*
	Method	if(*prop*.getDeclaredMethods().contains(*cnst*))	set *prop* to a Class instance that contains *cnst* Method
		if(*prop.name*==*cnst*)	set *prop* to a Class instance of *cnst*
	String	if(*prop.superClassName*==*cnst*)	set *prop* to a Sub-Class instance of *cnst*
		if(*prop.interfaceName*==*cnst*)	set *prop* to a Implementation Class instance of *cnst*
Method	Class	if(*prop.declaringClass*==*cnst*)	set *prop* to a Method instance of *cnst* Class
	String	if(*prop.name*.startWith(*cnst*))	set *prop* to a Method instance with proper name
	int	if(*prop.parameterNums*==0)	set *prop* to a Method instance with proper args numbers
Object	Class	if(*prop* instanceof *cnst*)	correct *prop* to an instance of *cnst*
Collection/Map	int	if(*prop*.size ≥ *cnst*)	add/remove elements
	Class	if(*prop.item* instanceof *cnst*)	correct the element type
	Object	if(prop.contains(cnst))	add/remove elements

to randomly select fields from *IOCD* that have not been marked with existing condition constraints for mutation.

(6) When the program reaches sinks and captures malicious instructions being written, that means the injection object is exploitable, and the corresponding gadget chain is classified as exploitable. If the time threshold is exceeded, or based on the specific type of information mentioned above, it can be directly determined as non-exploitable, the current test is terminated, and the next *IOCD* corresponding to the next gadget chain is invoked to start a new round of testing.

(ii) Mutation based on object structure constraints. After obtaining the condition constraints, JDD invokes the constraint solver to adjust the object's structure. Nevertheless, as demonstrated by IOCD, there exist intricate constraint relationships between fields. Modifying one of these fields can affect not only the corresponding ClassNode but also its fields and the fields associated with other ClassNodes. Consequently, a simplistic adjustment of an individual field based solely on condition constraints may compromise the structural validity of the object.

To address this issue, we propose our cascading mutation strategy. When JDD adjust a field, it cascadingly examines whether other fields affected by this field also require adjustments. If necessary, it adjusts them simultaneously to maintain the structural validity of the injection object. Specifically, the mutation strategy can be categorized into two levels: single-field and cross-field:

- Single-Field level. Mutating an individual field, which does not have any constraints with other fields. Utilizing constraint solvers to resolve relevant conditional constraints for adjusting the assignment of the field.
- Cross-Field level. When mutating a field with mutual constraints among other fields, ensure that relevant information in associated fields is preserved, and maintain the validity of the object structure.

(iii) Mutation based on field dependency constraints. Based on the insight of making minimal modifications to the affected fields to avoid disrupting valid information after mutation, and leveraging knowledge of class hierarchies, JDD has implemented the following two cross-field level mutators, i.e. nested field value reuse, sequential fixed adjustments.

- *Nested field value reuse*. After mutating a specific field (i.e., *mutatedField*), recursively check and adjust its associated fields. Specifically, start by adjusting the fields of *mutatedField* based on the relevant *dominator* constraints on the IOCD. Then, detect the fields affected by the current constraint strategy, and reuse instances shared between the post-mutation and pre-mutation field instances.
- *Sequential fixed adjustments*. If JDD intends to mutate a specific field (i.e., *field1*), and there exists a constraint associated with multiple fields (i.e., *field2*, *field3*), JDD extracts the constraints relevant to these three fields separately. It then calls a constraint solver to find values for *field1* that satisfy all related constraints while minimizing *field2* and *field3* adjustments. Specifically, the constraint solver first fixes *field2* and *field3* and then determines an appropriate value for *field1*. If it is unable to find a solution that satisfies the other constraints on *field1*, it progressively adjusts *field2* and *field3* and continues the solving process.

4. Implementation

We implemented JDD with over 25,000 lines of new Java code (excluding any third-party libraries). We will open-source JDD in the camera-ready version of the paper. The static taint analysis module of JDD is implemented based on Soot [11] and FlowDroid [12], a flow-sensitive, object-sensitive, and context-sensitive data-flow analysis tool. To identify the gadget fragments in dynamic proxies, we implemented a path-sensitive analysis inner their method invocation handlers. Moreover, we also conducted a lightweight pointer-to analysis to reduce the jump candidates in identifying gadget fragments. The dynamic fuzzing module of JDD is implemented based on JQF [13], [14]. We first use ASM [15] to instrument the target Java program for collecting the needed runtime context to guide our fuzzing engine. Then, we customized JQF to build a directional fuzzing framework for the generation of exploitable injection objects. To avoid JDD stuck in a hard-to-detect chain, we set

the time limitation for identifying each gadget fragment in the static analysis as 30 seconds and each round of fuzzing as 120 seconds, empirically.

5. Evaluation

In this section, we evaluate the performance of JDD on real-world Java programs and compare it with state-of-the-art tools via addressing four main research questions below:

- RQ1: How does JDD compare with state-of-the-art tools in detecting gadget chains?
- RQ2: How do the core modules of JDD contribute to its performance?
- RQ3: How many zero-day JOI-based RCE vulnerabilities can JDD detect?
- RQ4: How does JDD perform in analyzing real-world Java applications?

5.1. RQ1: Comparison with State of the Art

In this section, we compare JDD with state-of-the-art approaches in detecting (un)known gadget chains using two benchmarks. Specifically, we choose three state-of-the-art tools, namely GadgetInspector [3], SeHybrid [4], ODDFuzz [2]. To reduce randomness, we conducted five repetitions of each experiment and reported the average statistical results. All experiments were conducted on a Linux workstation with an Intel(R) Xeon(R) Gold 5218 CPU @ 2.30GHz and 128 GB of RAM, running Ubuntu 18.04 LTS.

Benchmarks. We conducted this experiment using a dataset of 61 well-known gadget chains that can be exploited to conduct JOI attacks. Among these gadget chains, 34 come from the well-known ysoserial repository [5], which has been widely used in prior work [2]–[4]. Additionally, we collected 26 other gadget chains by analyzing 32 recently disclosed JOI vulnerabilties. These vulnerabilities could cover popular deserialization protocols that are not included in ysoserial, for example, Hessian [16] and T3/IIOP [17] [18]. The new benchmark will be available together with our open-source repository.

Overall Results. Table 3 summarizes the overall results of the experiment. In this table, we present the quantities of gadget chains identified through static analysis (Identified chains), the number of gadget chains confirmed as exploitable (confirmed chains), and the number of new-discovered gadget chains beyond the scope of the known dataset (unknown chains). Note that, since SerHybrid, ODDFuzz, and JDD have dynamic validation modules, their confirmed chains come from the verified results of their dynamic analysis. For GadgetInspector, a tool that is solely reliant on static analysis, we manually verify the reported results to determine the number of confirmed chains.

Overall, for the capability of detecting known gadget chains, the effectiveness of JDD is higher than existing tools. Particularly, JDD covers 27 known and 91 (including two variants of known chains) unique unknown chains in ysoserial repository, which is much larger than GadgetInspector (three known + zero unknown), SerHybrid (two known + zero unknown), and ODDFuzz (16 known + zero unknown).

Note that JDD supports the identification and reuse of gadget fragments in known exploits. Since existing tools do not rely on known exploits, Table 3 does not include known exploits as input for the reason of fairness. We now describe the improvement if JDD is further improved by introducing the gadget fragments of known exploits. Take C3P0 as an example. If JDD is equipped with the known chain of C3P0, JDD can detect not only the gadget chains in Table 3 but also another five unknown gadget chains. Additionally, in the expanded dataset of 26 gadget chains extracted from recently disclosed JOI vulnerabilities, JDD also achieves a better result by detecting 24 known and 140 unknown chains.

False Positive Rates. The existing tools for static detection of gadget chains commonly exhibit a high false positive rate. For example, GadgetInspector reaches a false positive rate as high as 97.6%. However, SerHbrid, ODDFuzz, and JDD reduce the false positive rate of final results to 0% by introducing dynamic verification.

False Negative Rates. As shown in Table 3, JDD missed seven known gadget chains in ysoserial, resulting in a false negative rate of 20.6% (7/34). GadgetInspector and ODDFuzz have false negative rates of 91.2% (31/34) and 52.9% (18/34) respectively. SerHybrid could not support the test of all Java apps in ysoserial. On the dataset they tested, SerHybrid has a false negative rate of 83.3% (2/12). JDD demonstrated significantly lower rates of false negatives.

We then break down the seven known gadget chains that missed by JDD. First, four of them require specific domain knowledge to identify some of their gadget fragments. Specifically, JDD lacks domain knowledge of security-sensitive methods in Jython app, making it unable to detect the chain. Additionally, generating valid serialization data for the case in C3P0 requires rewriting a specific method. Moreover, JDD lacks the knowledge about how the return value of a dynamic proxy's invocation handler affects program execution, thus failing to detect two chains in Spring (i.e. Spring1,2). As aforementioned, we could use known exploits to further enhance JDD by identifying the gadget fragments used in them. For instance, JDD can automatically extract reusable fragments based on the exploit of Spring1 and further identify Spring2. Second, the rest three gadget chains contain many fields with a generic type, e.g., an Object type, leading to a time-out.

Benefits in Supporting Java Dynamic Features. Benefiting from the bottom-up fragment search strategy, JDD has the capacity to support more Java dynamic features to expand its detection capabilities, including *Polymorphic*, *dynamic proxy* and *reflection*, in which the last two are commonly unsupported by existing tools. As a result, JDD could detect more gadget chains related to these dynamic features than existing tools.

Table 3: Gadget chain detection comparison among GadgetInspector, SerHybrid, ODDFuzz, and JDD (Our Approach). The number in parentheses indicates the detected known chains in the benchmark.

Application	Known Chains	GadgetInspector		SerHybrid		ODDFuzz		JDD		
		Identified Chains	Confirmed Chains	Identified Chains	Confirmed Chains	Identified Chains	Confirmed Chains	Identified Chains	Confirmed Chains	Unkown Chains
Ysoserial Benchmark										
AspectJWeaver	1	8	0	N/A	N/A	9	0			
CommonsBeantuils	1	4	0	0	0	8	1	108†	25	20
CommonsCollections	5	4	1	1	1	97	3			
BeanShell	1	2	0	1	0	8	0	587	5	4
C3P0	1	2	0	N/A	N/A	13	1	15	0	0
Click	1	4	0	N/A	N/A	8	1	6	1	0
Clojure	1	12	1	N/A	N/A	184	1	6	3	2
CommonsCollections4	2	4	0	1	1	112	2	230	26	24
Groovy	1	4	0	3	0	13	0	413	5	4
JavassistWeld	1	2	0	N/A	N/A	8	0	6	1	0
JBossInterceptors	1	2	0	N/A	N/A	8	0	7	1	0
JDK	4	5	0	N/A	N/A	9	1	16	8	5
JSON	1	7	0	N/A	N/A	9	0	147	6	5
Jython	1	42	1	N/A	N/A	32	0	0	0	0
MozillaRhino	2	3	0	N/A	N/A	7	2	4	2	0
Hibernate	2	3	0	3	0	8	2	14	5	4
Myfaces	2	2	0	N/A	N/A	7	0	52	3	2
ROME	1	2	0	0	0	5	1	48	9	8
Spring	2	2	0	N/A	N/A	10	0	5	0	0
Vaadin	1	6	0	N/A	N/A	13	1	109	14	13
FileUpload	1	3	0	N/A	N/A	8	0	1	1	0
Wicket	1	3	0	N/A	N/A	7	0	1	1	0
Total	34	126	3 (3)	9	2 (2)	583	16 (16)	1362	116 (27)	91
Recently Disclosed Vulnerabilities										
Weblogic	21	53	0	N/A	N/A	N/A	N/A	642	126	107
MarshalSec(Hessian)‡	5	2	0	N/A	N/A	N/A	N/A	119	38	33
Total	26	55	0 (0)	-	-	-	-	761	164 (24)	140

† Due to the shared use of the CommonsCollections dependency in detecting Gadget Chains in AspectJWeaver and CommonsBeanutils, we simultaneously conducted analysis on these three packages.
‡ MarshalSec is a deserialization vulnerability exploitation tool, and we include its Hessian protocol-based Exploits as part of our evaluation dataset.

First, due to its support for the dynamic proxy feature, in our benchmark data set (i.e., the ysosearial repository), JDD can detect additional 16 gadget chains (3 known and 13 unknown chains) compared to existing tools.

Second, with the support of Java reflection, JDD can detect more variants of gadget chains. An example is Hibernate, which contains two variants of one unique chain in our benchmark. The difference of these two variants is in the last fragments, i.e, their targets of Java reflection. JDD successfully detects three unique chains in Hibernate and additionally identifies two fragments that could be linked as the targets of Java reflection to conduct code execution attacks. Another example is Vaadin. JDD first detects 14 gadget chains in it, which all end with Java reflection and JDD additionally identifies three fragments that could be linked to conduct code execution attacks.

5.2. RQ2: How do the core modules of JDD contribute to its performance?

We conducted two ablation studies on the task of gadget chain detection and exploitability verification by comparing JDD with the following variants to evaluate the effectiveness of JDD's different modules:

- JDD-top-down. We replace JDD's bottom-up strategy with top-down strategy proposed by previous tools, i.e. a Depth-first-search (DFS) starting from a source to a sink.
- JDD-NoIOCD. We remove JDD's *IOCD* guidance in the fuzzing module. Instead, JDD-NoIOCD employs an approach like ODDFuzz to infer the class hierarchy and randomly mutates fields based on a pre-defined dictionary.
- JDD-NoClassHierarchy: We replace JDD's class hierarchy inference approach with the approach proposed by previous tools, e.g., ODDFuzz.
- sys-NoConFd: We remove JDD's condition-constraints-aware and field-dependency-constraints-aware mutation. Because these two types of constraints are closely related, we remove them together. Additionally, JDD-NoConFd mutates fields randomly based on a pre-defined dictionary, similar to ODDFuzz.

First, we verify whether the bottom-up strategy is more effective than the top-down strategy, by comparing how many exploitable gadget chains can be found by these two methods. JDD completed the analysis in seven minutes and 58 seconds with no timeouts and detected 116

Table 4: The Detected Chains and Performance Evaluation Results of JDD on Real-World Java Apps.

Application	Basic Information			Detected Chains			Performance		
	Stars	Class Number	Method Number	Identified Chains	Confirmed Chains	Vendor Reply	Search and Link Gadget Chains	Construct IOCD	Dynamic Verify (Detected Chains)
Apache Dubbo	39K	88.5K	936.4K	31	7	CVE Assigned	8min29s	53s	36min15s (31)
Motan	6K	53.7K	454.7K	695	93	CVE Assigned	15min25s	47min13s	6h (358)
Solon	1.5K	280.9K	2,797.9K	117	35	CVE Assigned	39min56s	29min16s	2h35min59s (117)
XXL-Job	24.5k	52.5K	411.1K	843	110	CVE Assigning	7min24s	52min8s	6h (363)
Sofa-rpc	4.8K	94.9K	883.9K	205	43	CVE Assigned	37min15s	8min6s	3h43min3s (205)
Apache Tapestry	0.1K	28.8K	241.1K	16	5	CVE Assigning	54s	9s	19min38s (16)

exploitable gadget chains. In comparison, JDD-top-down took about 17 hours and only detected 15 gadget chains. After further analysis, we found that the analysis of 900 sources in JDD-top-down timed out, thus missing 101 exploitable gadget chains. For example, in the analysis of `CommonsCollections4` app, JDD-top-down failed to complete the analysis of `PriorityQueue.readObject` within the time threshold (e.g. five minutes). As a result, JDD-top-down missed two exploitable chains.

Second, we evaluate the effectiveness of three types of guidance information in dynamic verification - class hierarchy constraints, conditional constraints and field dependency constraints, as well as the effectiveness of dataflow-based class hierarchy inference approach, by comparing the detection capabilities of JDD with JDD-NoIOCD, JDD-NoClassHierarchy and JDD-NoConFd on exploitable gadget chains.

Experimental results show that JDD performs best, with a total of 116 exploitable gadget chains detected, far exceeding JDD-NoIOCD (31), JDD-NoClassHierarchy (41) and JDD-NoConFd (38). The experimental results prove that to generate exploitable injection objects for gadget chains efficiently, the three types of constraint information in IOCD (described in Section 3.5) are essential, especially for complex chains. For example, for the chain in our motivating example, JDD-NoClassHierarchy and JDD-NoIOCD will infer the wrong class hierarchy as shown in Figure 3 (b), causing subsequent mutations to be meaningless. Although JDD-NoConFd can generate seeds with the correct class hierarchy, it is challenging to generate an object that simultaneously satisfies the constraints (e.g. i-iii in Figure 3) within the time threshold, so this case is easily missed.

In addition, compared with ODDFuzz, JDD-NoIOCD detected 15 more exploitable chains on the same benchmark ysoserial. This is because JDD 's static analysis module can provide more high-quality candidate chains, which also means that JDD 's static analysis module can help to improve ODDFuzz's detection capabilities.

5.3. RQ3: Discovering Zero-day Vulnerabilities

In this section, we run JDD atop real-world popular Java applications to detect zero-day JOI-based RCE vulnerabilities. Specifically, we first collected 80 popular Java open-source apps from Github [19] with more than 100 stars. Then, we implement a semi-automatic tool to check if they contain public entries for Java object injection and deserialization. As a result, we successfully recognized six apps (shown in Table 4) and use them as the dataset of this experiment. For these six popular Java apps, we conducted our analysis based on the default or recommended configuration within their open-source projects. In this experiment, the total time limitation for dynamic testing is 6 hours.

Table 4 shows the results, which indicates that JDD successfully identified zero-day JOI vulnerabilities in all of these six Java apps as well as generated exploitable injection objects for the 293 found gadget chains (127 unique chains). We have reported these gadget chains to the app developers and received confirmation from them. For example, the developers of Dubbo admitted that it is impossible for them to uncover these gadget chains, which indicates the urgency of our automatic tool. All 127 zero-day gadget chains detected by JDD are capable of bypassing the latest patch defenses at the time and, hence are vulnerabilities. Noting that, since we reported multiple gadget chains for each attack point to help developers design and implement more comprehensive defenses, we ultimately obtained six CVEs, less than the total count of chains.

To demonstrate how our fragment based bottom-up approach can benefit the detecting of gadget chains, we use real-world zero-day vulnerabilities found by JDD for case study.

Case Study #1: JOI based RCE Attacks. JDD discovered a JOI based RCE vulnerability that impacts many popular Java apps such as Sofa, Solon, and XXL-Job, in which XXL-Job is a highly popular distributed task scheduling framework with a remarkable number of GitHub stars (24.9k) and forks (10.3k) as well as be widely used by large enterprises with over 256 million consumers. This vulnerability could allow attackers to take over the victim server by sending a request with a well-crafted injection object.

Thses application utilize the Hessian/Hessian-sofa protocol to deserialize the received serialized data. The main attack process for this vulnerability is illustrated in Figure 5, wherein the attacker initializes a `HashMap` object and places two other `HashMap` objects (i.e. *hmap1*, *hmap2*) inside it, as well as ensures that *hmap1*, *hmap2* have identical hash values to trigger the jump from Fragment 1 to 2. Then, to facilitate the jump from Fragment 2 to Fragment 3 and then to 4, the attacker needs to insert a `Type` instance and a `JSONObject` instance into *hmap1* and *hmap2*, and alternate the order of insertion. Then, based

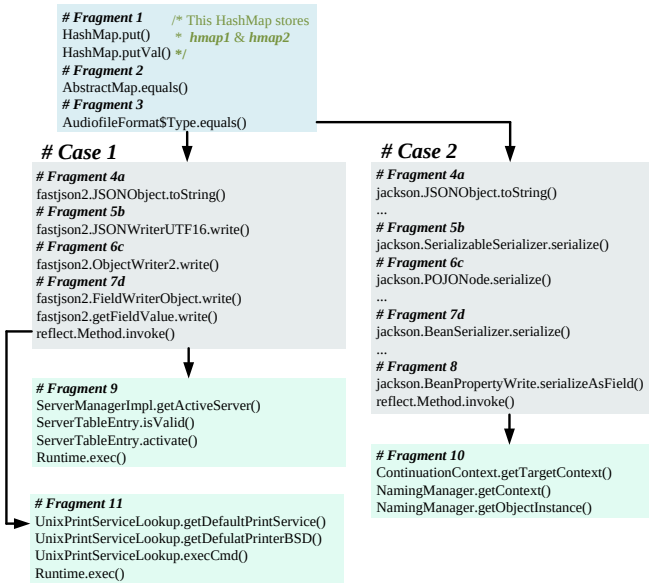

Figure 5: Simplified Gadget Chains of Case Study #1 and Case Study #2, where Case #2 is an evolution of Case #1 with some gadget fragments being replaced.

on the information about the object (i.e., *obj*) contained in the `JSONObject` instance, the deserialization process will jump from Fragment 4a to 7d, and subsequently invoke reflective calls to methods related to the fields of *obj*, such as "getDefaultPrintService()" method of a `UnixPrintServiceLookup` object. This method further leads to the execution of arbitrary commands injected by the attacker through `Runtime.exec()`. Alternatively, as a variant, this gadget chain also can utilize Java reflection in Fragment 7d to invoke `ContinuationContext.getTargetContext()` and carry out JNDI based code injection attacks.

Case Study #2: The Evolution of Case Study #1. By replacing certain fragments in a gadget chain, an attacker can rapidly discover new gadget chains that can affect another app. Taking the gadget chain introduced in Case Study 1 as an example, this chain can affect apps like Sofa-RPC, solon, XXL-Job. However, due to the absence of the required Java library dependency in Motan by default, this gadget chain cannot be used to exploit Motan, as some of its gadget fragments missed.

However, if the missed fragments could be replaced by gadget fragments in Motan, it could generate a new chain. Actually, in Motan, `JDD` detected an equivalent fragment that depends on Jackson [20], a widely-used third-party dependency for data-binding functionality and tree-model. Thus, the Fragment 4a-8 (in # Case 2) can replace the Fragment 4a-7d (in # Case 1), resulting a new chain, as illustrated in Figure 5.

Furthermore, the gadget chain in Case Study 1 uses the "getDefaultPrintService()" method of a `UnixPrintServiceLookup` object to load and execute code from remote attackers. However, this method could only be invoked in a Unix-like system. For those non-Unix systems, `JDD` could further find alternative fragments to replace it thus fixing the chain, for example, using "getActiveServers()" method in `ServerManagerImpl` class.

5.4. RQ4: Performance

In this section, we evaluate the performance of `JDD` in analyzing real-world Java apps. Table 4 shows the results. For the easy of statistic, we separate the analysis time of `JDD` into three parts: (i) the gadget chain searching and linking (i.e., Steps 1–3 in Section 3), (ii) IOCD construction (i.e., Step 4 in Section 3), and (iii) directional Fuzzing (i.e., Step 5 in Section 3). The results indicate that `JDD` could finish its analysis for most of real-world apps in the time limitation (six hours for each app).

We have two observations. First, the static analysis part of `JDD` (Steps I–III) is very efficient, finishing with 20 mins in most cases. The reason is that `JDD`'s static search time is polynomial, i.e., efficient in finding possible gadget chains. The analysis of Solon is slower but still within 40 minutes because the number of possible gadget fragments is large. Second, the dynamic analysis part of `JDD` may still have performance issues. Specifically, the analysis of Motan and Sofa-rpc times out after six hours due to false positives in gadget chains reported by `JDD`'s static analysis. The main reason is the imprecision in our points-to analysis and path insensitivity outside invocation handlers of dynamic proxy. We leave the improvement of static analysis's precision as our future work.

6. Discussion

Expandability for Incomplete Patching Problem. One common problem facing JOI vulnerabilities and their patches is that remote attackers may find alternatives for the blocked gadgets in the patch and derive new exploitable gadget chains from the original ones. For example, Figure 6 illustrates the gadget chains' reuse relationships among 23 JOI vulnerabilities that affect WebLogic [21], a widely-used J2EE application server employed by over 430K Java applications. Notably, CVE-2015-2852 continuously evaded security patches by successively replacing parts of the fragments that were blacklisted, leading to a total of 11 follow-up CVEs over a period of six years.

There are two methods for users to expand `JDD` for the incomplete patching problem. First, users may expand the fragment data set of `JDD` with known fragments, e.g., all possible gadgets in CVE-2015-2852. Second, users may expand the source set of `JDD` with known sources. These two methods will help users to find possible derivative vulnerabilities, thus making the patch more complete.

Lightweight Pointer-to Analysis and Restricted Path-sensitive Analysis. We leverage pointer-to analysis and path-sensitive analysis to mitigate the erroneous generation

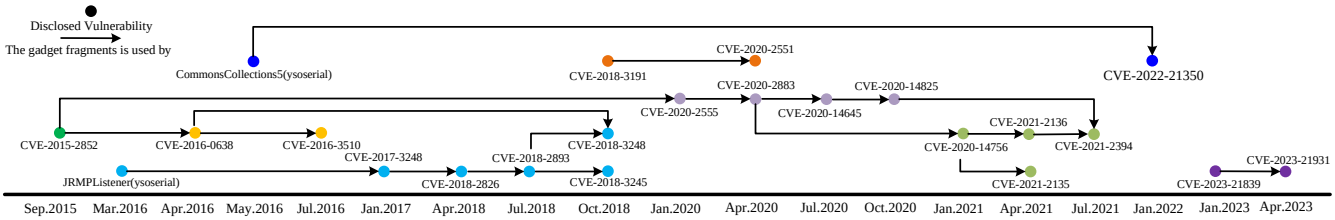

Figure 6: WebLogic's JOI vulnerabilities and the reuse of their gadget fragments, which lead to the incomplete patch problem.

and concatenation of fragments, thereby reducing false positives in static analysis and lowering the computational load of fuzzing.

The misidentification of an object's type can result in the erroneous identification of dynamic method invocations, consequently leading to false positives in fragment generation and linking. The utilization of pointer-to analysis techniques effectively reduces false positives caused by this issue. For instance, in the analysis of the `CommonCollections4` application, applying the lightweight pointer-to analysis results in a reduction of 806 false positive chains.

Furthermore, dynamic proxies are widely used in many Java apps. Specifically, an invocation handler of dynamic proxy usually selects different program handling logic based on the method properties (e.g., method name) that triggers the handler, resulting in many possible execution paths during static analysis. Thus, it requires a path-sensitive analysis to identify accurate execution paths when linking gadget fragments in invocation handlers of dynamic proxies. But applying the path-sensitive analysis to a larger scope (e.g., the entire codebase) will significantly slow down static analysis with limited gains in reducing false positives, so we limit the scope of path-sensitive analysis to balance performance and false positives.

Limitations. We discuss two limitations of JDD: (i) performance of dynamic analysis, and (ii) implicit constraints. First, while JDD significantly improves the performance of static analysis, the fuzzing part of dynamic analysis may still have performance issues during mutation, particularly when the number of false positives from static analysis is large for some Java applications due to over-approximation. The main reasons for such false positives are that our points-to analysis is lightweight, leading to imprecisions in many cases, and our path sensitivity is restricted inside invocation handlers of dynamic proxies, i.e., the analysis outside is path insensitive. Second, while JDD considers some implicit constraints, such as the nullpointer exceptions and forced type casting, there are still other implicit constraints that need to be solved during exploit generation.

7. Related Work

We describe both attacks and defenses related to deserialization vulnerabilities.

7.1. Deserialization Vulnerability Mining

In recent years, a substantial body of research [22]–[24] has been devoted to the automated mining of deserialization vulnerabilities using both static and dynamic methods.

GadgetInspector [3] is one of the most well-known automated gadget chain mining tools. It utilized static taint analysis to discover execution paths from manually defined source points to sink points. However, its effectiveness is limited in intricate scenarios due to inherent design limitations in data-flow and control-flow analysis. Chen et al. proposed Tabby [25], which utilizes Soot [11] to transform Java programs into Code Property Graphs (CPG) [26]–[29] and imports them into Neo4j database [30]. Then, they extract gadget chains through querying CPG, using manually crafted Cypher [31] statements. These approaches commonly demand significant expertise in deserialization vulnerabilities, thus constraining their scalability. Compared with them, during static analysis, JDD do not rely on domain knowledge about JOI vulnerabilities. It only leverages the understanding of Java dynamic features to identify gadget fragments and link them together. Moreover, JDD also designs a novel bottom-up approach to greatly improve the efficiency of its static module.

Considering that static analysis often introduces numerous false positives, SerHybrid [4] proposed to automate the verification of exploitable gadget chains through the construction of heap abstractions for generating injection objects. Building upon Tabby, GCMiner [1] incorporated an overriding-guided object generation strategy to automate verification during fuzzing. And ODDFuzz [2] utilized directed fuzzing to enhance verification efficiency. In addition, black-box scanning tools like Marshalsec [32] and Java Deserialization Scanner [33] have also been proposed. Compared with them, JDD guides fuzzing by recognizing data-flow dependencies between injection objects' fields. On one hand, this approach enables the inference of complex injection object structures, such as parallel and embedded injection object, which were commonly uncovered in previous work. Furthermore, JDD significantly improves seed mutation's efficiency by considering constraints from the program's execution flow specified by gadget chains. It is noteworthy that, to the best of our knowledge, JDD is the first tool capable of automatic generation of exploitable injection objects based on gadget chains.

7.2. Deserialization Vulnerability Defense

Implementing effective strategies to enhance application security is often necessary, particularly in the context of defense against deserialization vulnerabilities [34]. Existing strategies are commonly referred to as look-ahead defenses [35], wherein an audit is conducted on classes that require deserialization, and only those that are permitted can be deserialized. One notable look-ahead defense is JEP290 [36], which offers a blacklist of prohibited malicious classes during deserialization, along with interfaces for user extensions. In addition to the JDK's built-in standard solutions, Runtime Application Self-Protection (RASP) [37] frameworks [38] [39] are frequently employed to defend against deserialization vulnerabilities. These frameworks leverage bytecode instrumentation tools to intercept risky APIs for auditing. Some work also concentrate on automating the generation of deserialization strategies. Trusted [40] is a two-phase defense framework that learns from benign deserialization workloads to create an allowlist, which is then enforced during the deserialization process. Another approach by François et al. [41] involves training a Markov chain with malicious Gadget Chains, which is then used to predict malicious Injection Objects at runtime.

In summary, this thread of work commonly enforce security checks on risky sources, sinks, and gadgets used in Java deserialization. Compared with them, JDD focuses on another important research problem – mining risky sources, sinks, and gadgets that could be abused as much as possible. Thus, the output of JDD can significantly support them. Take the white list adopted by Trusted as an example. JDD could check if the classes in the white list are enough to construct gadget chains. Another example is RASP based approach. Their efficiencies highly depend on the completeness of risky sink points, and JDD could help to verify if specific methods could be exploited by injection objects and thus should be protected.

8. Conclusion

In this paper, we design a novel, scalable approach for automated detection and verification of JOI vulnerabilities, called JDD. On one hand, JDD solves the static path explosion problem by a bottom-up approach, which first looks for gadget chain fragments and then chains gadget fragments from sinks to sources. The approach reduces static search time from exponential to polynomial, i.e., from $O(eM^n)$ to $O(M^2n^3 + enM)$, where n is the number of dynamic function calls in a gadget chain, M is the average number of dynamic function call candidates, and e is the number of entry points. On the other hand, JDD constructs a so-called Injection Object Construction Diagram (IOCD), which models the data-flow dependencies between injection objects' fields to facilitate dynamic fuzzing. Our evaluation of JDD upon six real-world Java applications reveals 127 zero-day, exploitable gadget chains with six being assigned with Common Vulnerabilities and Exposures (CVE) identifiers. We also responsibly reported these vulnerabilities to application developers and obtained their acknowledgments and confirmations.

Acknowledgments

We would like to thank the anonymous reviewers for their insightful comments that helped improve the quality of the paper. This work was supported in part by the National Key Research and Development Program (2021YFB3101200), National Natural Science Foundation of China (62172104, 62172105, 61972099, 62102093, 62102091), and National Science Foundation (NSF) (CNS-21-54404 and CNS-20-46361). Yuan Zhang was supported in part by the Shanghai Rising-Star Program under Grant 21QA1400700 and the Shanghai Pilot Program for Basic Research - FuDan University 21TQ1400100 (21TQ012). Dr. Yinzhi Cao was supported in part by Johns Hopkins Catalyst Awards and DARPA Young Faculty Award under Grant Agreement D22AP00137-00 as well as a gift from Visa Research. Min Yang is the corresponding author, and a faculty of Shanghai Institute of Intelligent Electronics & Systems and Engineering Research Center of Cyber Security Auditing and Monitoring.

References

[1] 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 IEEE/ACM 45th International Conference on Software Engineering (ICSE), 2023, pp. 397–409.

[2] S. Cao, B. He, X. Sun, Y. Ouyang, C. Zhang, X. Wu, T. Su, L. Bo, B. Li, C. Ma, J. Li, and T. Wei, "Oddfuzz: Discovering java deserialization vulnerabilities via structure-aware directed greybox fuzzing," in IEEE Symposium on Security and Privacy (SP), 2023, pp. 2726–2743.

[3] I. Haken, "Automated discovery of deserialization gadget chains," Proceedings of the Black Hat USA, vol. 48, 2018.

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

[5] Ysoserial. https://github.com/frohoff/ysoserial.

[6] Sofa-rpc. https://github.com/sofastack/sofa-rpc.

[7] https://github.com/apache/dubbo-hessian-lite.

[8] https://github.com/sofastack/sofa-hessian.

[9] 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, 2018, pp. 209–220.

[10] Y. Li, T. Tan, and J. Xue, "Understanding and analyzing java reflection," ACM Transactions on Software Engineering and Methodology (TOSEM), vol. 28, no. 2, pp. 1–50, 2019.

[11] Soot. https://github.com/soot-oss/soot.

[12] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," Acm Sigplan Notices, vol. 49, no. 6, pp. 259–269, 2014.

[13] 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, 2019, pp. 398–401.

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

[15] Asm. https://asm.ow2.io/.

[16] Hessian protocol. https://en.wikipedia.org/wiki/Hessian_(Web_service_protocol).

[17] T3 protocol. https://docs.oracle.com/cd/E14571_01/web.1111/e13721/rmi_t3.htm#WLRMI143.

[18] Iiop protocol. https://www.ibm.com/docs/en/iad/7.2.1?topic=i-internet-inter-orb-protocol-iiop.

[19] Github. https://github.com/.

[20] Jackson. https://github.com/FasterXML/jackson.

[21] Oracle weblogic server. https://www.oracle.com/java/weblogic/.

[22] S. Park, D. Kim, S. Jana, and S. Son, "{FUGIO}: Automatic exploit generation for {PHP} object injection vulnerabilities," in 31st USENIX Security Symposium (USENIX Security 22), 2022, pp. 197–214.

[23] M. Shcherbakov and M. Balliu, "Serialdetector: Principled and practical exploration of object injection vulnerabilities for the web," in Network and Distributed Systems Security (NDSS) Symposium 202121-24 February 2021, 2021.

[24] S. Cristalli, E. Vignati, D. Bruschi, and A. Lanzi, "Trusted execution path for protecting java applications against deserialization of untrusted data," in Research in Attacks, Intrusions, and Defenses: 21st International Symposium, RAID 2018, Heraklion, Crete, Greece, September 10-12, 2018, Proceedings 21. Springer, 2018, pp. 445–464.

[25] Z. J. Y. F. X. L. X. F. X. Chen, B. Wang and Q. Liu, "Tabby: Automated gadget chain detection for java deserialization vulnerabilities," 2023.

[26] "Code property graph - Wikipedia — en.wikipedia.org," https://en.wikipedia.org/wiki/Code_property_graph, [Accessed 04-08-2023].

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

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

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

[30] neo4j. https://neo4j.com/.

[31] Cypher. http://opencypher.org/.

[32] marshalsec. https://github.com/mbechler/marshalsec.

[33] Java deserializer sanncer. https://github.com/federicodotta/Java-Deserialization-Scanner.

[34] "CWE - CWE-502: Deserialization of Untrusted Data (4.12) — cwe.mitre.org," https://cwe.mitre.org/data/definitions/502.html, [Accessed 04-08-2023].

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

[36] jeps290. https://openjdk.org/jeps/290.

[37] Z. L. Z. Yin and Y. Cao, "A web application runtime application self-protection scheme against script injection attacks," p. 566–577, 2018.

[38] openrasp. https://github.com/baidu/openrasp.

[39] P. Čisar and S. M. Čisar, "The framework of runtime application self-protection technology," in 2016 IEEE 17th International Symposium on Computational Intelligence and Informatics (CINTI), 2016, pp. 000 081–000 086.

[40] D. B. S. Cristall, E. Vignati and A. Lanzi, "Trusted execution path for protecting java applications against deserialization of untrusted data," Research in Attacks, Intrusions, and Defenses, M. Bailey, T. Holz, M. Stamatogiannakis, and S. Ioannidis, Eds. Cham: Springer International Publishing, pp. 445–464, 2018.

[41] F. Gauthier and S. Bae, "Runtime prevention of deserialization attacks," in Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results. New York, NY, USA: Association for Computing Machinery, 2022, p. 71–75. [Online]. Available: https://doi.org/10.1145/3510455.3512786

[42] "Cauchy–Schwarz inequality - Wikipedia — en.wikipedia.org," https://en.wikipedia.org/wiki/Cauchy%E2%80%93Schwarz_inequality, [Accessed 04-08-2023].

Appendix A.
Time Complexity of Bottom-up and Top-down Approaches

A.1. Proof of Theorem 1

Proof. If there are n dynamic method invocations in a chain, and the average number of candidates for these invocations is M, the number of *source* is e. According to Lemma 1, the search complexity for a source is $O(n^3M^2)$.

As for each source, the maximum number of connection checks is nM, and the overall search complexity of Algorithm 1 to search for all sources is $O(n^3M^2 + enM)$. □

Lemma 1. *The per-entry search complexity of Algorithm 1 is $O(M^2n^3)$, where n is the number of dynamic method invocations, and M is the average number of candidates for these invocations.*

Proof. In Algorithm 1, n' represents the number of distinct dynamic method invocations, with each dynamic method having x_1, x_2, ..., $x_{n'}$, candidates, as difference groups, and a total of $N = x_1 + ... + x_{n'}$ candidates. We can then compute the searching complexity for Algorithm 1 through the following three steps.

- **Max Searching Times per Round**: By employing a bottom-up search, it becomes possible to store the exact input requirements connecting each fragment to *Sink Fragments* and avoid repeated searching. Meanwhile, there will not be two *Sink Fragments* with the same dynamic method implementation serving as the *Head*. And the fragment with the candidate of the p-th dynamic method invocation as the *Head* requires checking for connections with a maximum of $N - x_p$ other fragments. Considering these most complex scenarios, using equations (1) and (2), we can calculate

the maximum number S of linkage detection attempts during the search process.

$$x_1 + x_2 + ... + x_{n'} = N <= Mn \quad (1)$$

$$S_p = x_p(N - x_p) \quad (2)$$

$$S = N^2 - (x_1^2 + ...x_{n'}^2) \quad (3)$$

According to the Cauchy-Schwarz inequality [42], for any real numbers $a_1, a_2, ..., a_n$ and $b_1, b_2, ..., b_n$, we have:

$$(a_1^2 + a_2^2 + ... + a_{n'}^2)(b_1^2 + b_2^2 + ... + b_{n'}^2) >= (a_1b_1 + a_2b_2 + ... + a_{n'}b_{n'})^2 \quad (4)$$

When we set $a_1 = x_1, a_2 = x_2, ..., a_{n'} = x_{n'}$, and $b_1 = b_2 = ... = b_{n'} = 1$, equation (4) is transformed into (5),

$$(x_1^2 + x_2^2 + ... + x_{n'}^2) >= \frac{(x_1 + x_2 + ... + x_{n'})^2}{n'} = \frac{N^2}{n'} \quad (5)$$

From equations (5) and (3), it follows that

$$S <= N^2(1 - \frac{1}{n'}) \quad (6)$$

, and the maximum value of S is obtained when $x_1 = x_2 = ... = x_n = \frac{N}{n'}$.

- **Searching Complexity** Since the search requires a maximum of n' iterations, the overall algorithm's maximum number of search attempts is

$$n'N^2(1 - \frac{1}{n'}) < n'N^2 <= n^3M^2 \quad (7)$$

We define the search complexity $O(1)$ as a one-time fragment linking check. Note that, after `JDD` identifies the fragments, it will merge the fragments whose headers are the implementations of the same dynamic method but meet the conditions based on the special cases described in Appendix B. The complexity of this step is O(N); Therefore, the final maximum search complexity is $O(n^3M^2 + N) = O(n^3M^2)$ for each *source*.

□

A.2. Proof of Theorem 2

Proof. All symbols follow the Theorem 1. Based on the description in the ODDFuzz paper, when an attacker-controllable dynamic method call is searched, all overridden methods will continue to be searched. Therefore, under the assumption of a search strategy that avoids false negatives as much as possible, and also considers the computational complexity in the worst case, conducting a DFS (Depth-First Searching) for potential gadget chains from a *source* to a *sink* can be equivalent to detecting all paths of a tree with depth n, and each node in the tree having an average of M branches, starting from the root node, using the DFS algorithm. The computational complexity of this process can be determined accordingly.

- The notation O(1) is defined as follows: when the tree is an empty tree (has no nodes) or consists of only one node, the number of paths is 0 or 1 respectively. This serves as the termination condition for the recursion. The search complexity is denoted by S(1) = O(1).
- **The recursive hypothesis.** Assuming that for a node at a depth d in the tree, the searching complexity of the Depth-First Search (DFS) algorithm is denoted by $S(d)$.
- **The recursive step:** For a node at depth $d - 1$, we recursively invoke the DFS function to process its child nodes. Each node can have M child nodes, so in the worst case, the DFS function needs to be called M times for each child node located at depth $d - 1$. According to the induction hypothesis, the DFS processes nodes at depth $d-1$ with a searching complexity of S(d-1). Therefore, the time complexity to process a node at depth d is $O(M) \times S(d-1)$.
  Based on the recursive relation, we can expand

$$\begin{aligned} S(d) &= O(M) \times S(d-1) \\ &= O(M^2) \times S(d-2) \\ &= ... \\ &= O(M^d) \times S(0) \\ &= O(M^d) \end{aligned}$$

Therefore, for a single source, the search complexity is $O(M^n)$, and when performing DFS searches from e entry points, the search complexity becomes $O(eM^n)$.

□

Appendix B. An analysis of historical gadget chains

We collected 79 gadget chains from well-known deserialization vulnerability databases, including ysoserial, MarshalSec, and 113 CVE reports of JOI vulnerabilities.

When studying these 79 gadget chains, we first explored which methods are typically chained after Java reflection. The results show that 39 of the 79 gadget chains use Java reflection, and they can all be summarized into two major types. Specifically, 27 chains use parameterless methods as follow-up methods. Of these 27 chains, six require *getter* or *is* methods and four must be overridden methods of interface methods. The remaining 12 chains use methods with a parameter of type String or Boolean as follow-up methods to unsafe reflection. Therefore, we prioritize testing the two types of successor methods in Section 3.3.

Then, among these 79 historical gadget chains, only nine have the same dynamic method calls nested, and the reasons for nesting the same dynamic method calls can be attributed to two categories: (1) Implement chain calls of `Method.invoke`, sequentially calling insecure methods in

the program to execute malicious instructions, such as CVE-2020-2555. (2) Satisfy common specific strong constraints - hash collision, such as the conditional constraints shown in Line 7 in Figure 1. Therefore, we let JDD support both.

Appendix C. Meta-Review

The following meta-review was prepared by the program committee for the 2024 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

C.1. Summary

This paper proposes JDD, a new framework to detect Java deserialization vulnerabilities. The main focus in on improving the efficiency of the analysis. First, it mitigate the path explosion problem by employing a bottom-up search strategy to avoid redundancies in top-down approaches, which reduce the time needed for static gadget chain search from exponential time to polynomial time. Second, it uses the Injection Object Construction Diagram (IOCD) to track dependencies between data fields of the various injected objects, which reduces the search scope. Third, it tracks control dependencies during fuzzing to collect more fine-grained progress feedback. Evaluation shows that the prototype tool is able to find more deserialization gadget chains than previous state-of-the-art approaches.

C.2. Scientific Contributions

- Provides a New Data Set For Public Use
- Creates a New Tool to Enable Future Science
- Provides a Valuable Step Forward in an Established Field

C.3. Reasons for Acceptance

1) The paper is well-written
2) The paper tackles with an interesting and timely topic: deserialization vulnerabilities in Java
3) JDD discovered 127 unknown deserialization chains in 6 real-world Java apps
4) JDD outperformed the previous state-of-the-art, in terms of scalability and false negative rate