Gleipner: A Benchmark for Gadget Chain Detection in Java Deserialization Vulnerabilities

BRUNO KREYSSIG, Umeå University, Sweden
ALEXANDRE BARTEL, Umeå University, Sweden

While multiple recent publications on detecting Java Deserialization Vulnerabilities highlight an increasing relevance of the topic, until now no proper benchmark has been established to evaluate the individual approaches. Hence, it has become increasingly difficult to show improvements over previous tools and trade-offs that were made. In this work, we synthesize the main challenges in gadget chain detection. More specifically, this unveils the constraints program analysis faces in the context of gadget chain detection. From there, we develop *Gleipner*: the first synthetic, large-scale and systematic benchmark to validate the effectiveness of algorithms for detecting gadget chains in the Java programming language. We then benchmark **seven** previous publications in the field using *Gleipner*. As a result, it shows, that (1) our benchmark provides a transparent, qualitative, and sound measurement for the maturity of gadget chain detecting tools, (2) *Gleipner* alleviates severe benchmarking flaws which were previously common in the field and (3) state-of-the-art tools still struggle with most challenges in gadget chain detection.

CCS Concepts: • **Security and privacy → Software and application security**; • **Software and its engineering → Software defect analysis**; *Software libraries and repositories*; *Object oriented development*.

Additional Key Words and Phrases: deserialization, benchmark, gadget chain, Java, vulnerabilitiy, program analysis

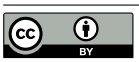
ACM Reference Format:
Bruno Kreyssig and Alexandre Bartel. 2025. Gleipner: A Benchmark for Gadget Chain Detection in Java Deserialization Vulnerabilities. *Proc. ACM Softw. Eng.* 2, FSE, Article FSE001 (July 2025), 21 pages. https://doi.org/10.1145/3715711

1 Introduction

Insecure Deserialization is one of the *OWASP Top Ten* most severe vulnerabilities in software [22]. It first gained significant attention with the talk from *Frohoff et al.* in 2015 [25] and has since then inspired further research into its intricacies [2, 30, 50], mitigation [26, 46, 56, 59], and detection techniques. The latter can be further divided into detecting insecure deserialization entry points [16, 20, 29] and finding so-called gadget chains within source code or bytecode [3, 9, 12, 14, 15, 27, 31, 35, 36, 44, 52, 58]. Indeed, both conditions must be met to result in a deserialization vulnerability. In this work, we focus on the challenge of detecting deserialization gadget chains in Java, as it lately received abundant attention, and so far, no proper benchmark has been established to verify the individual approaches.

The idea of assessing vulnerability detection tools is not novel in itself. Beginning with the *Juliet* test suite for Java and C/C++ (2010) [7], large datasets of both synthetic and real-world vulnerable software artifacts were constructed to cover test scenarios for a variety of security-related products. For instance, there are numerous benchmarks for automated program repair (APR) [4, 10] and

Authors' Contact Information: Bruno Kreyssig, Umeå University, Umeå, Sweden, bruno.kreyssig@umu.se; Alexandre Bartel, Umeå University, Umeå, Sweden, alexandre.bartel@cs.umu.se.



ACM 2994-970X/2025/7-ARTFSE001
https://doi.org/10.1145/3715711

general web application assessment (in particular, the OWASP Benchmark [41]). Furthermore, the NIST *Software Assurance Reference Dataset* (SARD) [5] aims to centrally collect labeled program weaknesses. At the time of writing, SARD contains **46 447** test cases for weaknesses in Java applications. Yet, only one test case is related to CWE-502 [38], the common weakness identifier for insecure deserialization. In addition, this test case describes a trivial example based on directly tainting request fields instead of constructing a specific gadget chain. *Sabatini* created a benchmark for testing generic *Static Application Security Testing* (SAST) tools whether they are capable of detecting common patterns of insecure deserialization in Java (i.e., a call to *readObject()*) [47]. Again, this benchmark only considers insecure deserialization entry points.

The artifact closest to resembling a benchmark for Java deserialization gadget chains is *Ysoserial* [24]. By design, this tool is a payload generator for 46 [30] known gadget chains within the Java Class Library and third-party dependencies. Given no alternatives, the evaluation of previous gadget chain detection tools has relied on taking these dependencies, known to contain gadget chains as per the *Ysoserial* repository, and determining whether their tool also finds precisely these gadget chains. However, testing tools against a limited set of real-world dependencies as a means of evaluation is inefficient, incomplete, and lacks transparency. These issues manifest themselves in common benchmarking malpractices [55]:

(1) **Lack of false postives and negatives:** the true number of gadget chains that exist within a dependency is unknown. While it is laborious to manually verify all findings reported by a tool to determine the amount of false positives found, false negatives are only visible for the very small ground truth of gadget chains known from the *Ysoserial* repository. That is often a single gadget chain within a combination of dependencies (e.g., the Ysoserial gadget chains JSON1, JavassistWeld, Hibernate [24]).

(2) **Performance degradation:** while tools may find more gadget chains in the limited set of dependencies in *Ysoserial*, it is impossible to determine the trade-offs made to improve the tool and how this might negatively impact the gadget chain detection tool in other scenarios.

(3) **Selective datasets:** for example, the publication *Tabby* [15] takes the dependencies of 36 gadget chains in *Ysoserial* and two further dependencies from the no longer maintained *MarshalSec* repository [2]. Then, *Crystallizer* [52] takes only seven Java dependencies into account, and further, *JChainz* [9] and *HawkGadget* [58] are evaluated solely against the *Apache Commons Collections* dependency.

(4) **Not benchmarking against the state-of-the-art:** recent publications in top-tier venues (e.g., *Crystallizer* [52], *Tabby* [15], *JDD* [14]) mostly rely on comparison of their tool against three dated tools: *Serianalyzer* [3] ('17), *GadgetInspector* [27] ('18) and *SerHybrid* [44] ('20).

(5) **Failing to evaluate all claimed contributions:** e.g., the recent publication *JDD* claims to improve gadget chain detection under the constraints of dynamic Java features such as reflection and dynamic proxies [14]. It is not self-evident from showing to have found gadget chains within a dependency that, thereby, these challenges were sufficiently tackled.

To counteract these malpractices, we build *Gleipner*[1], a synthetic benchmark dataset, which by design ensures a solid ground truth of true positive and false negative Java deserialization gadget chains. These chains represent both fine-grained challenges known to gadget chain detection tools and Java program analysis in general, as well as synthetic replicas of real-world gadget chains from *Ysoserial*. We then evaluate all **seven** available gadget chain detection tools against our benchmark. As such, our main contributions are:

- A summary of challenges in gadget chain detection and Java program analysis as a whole.

[1]In Norse mythology, *Gleipner* are the chains that bound the mighty wolf *Fenrir*. https://en.wikipedia.org/wiki/Gleipnir

- The synthesis of these challenges into granular test-cases to evaluate the performance of gadget chain detection tools on these challenges in isolation.
- The re-implementation of 12 real-world gadget chains from *Ysoserial*
- The design and implementation of *Gleipner*, the first benchmark designated for evaluating Java gadget chain detection algorithms.
- A systematic evaluation of all available gadget chain detection tools on our benchmark.

The remainder of the paper is structured as follows. In Section 2, we define the concept of gadget chains in Java deserialization vulnerabilities and provide a motivating example to illustrate the complexity of the problem. Then, we analyze the challenges gadget chain detection tools face. We use this knowledge in Section 3 to design and implement our benchmark, which we verify on all currently available tools in Section 4. This opens the discussion on how *Gleipner* improves the evaluation of gadget chain detection tools (Section 5). Finally, we present the related work in Section 6 and the limitations in Section 7.

2 Background

2.1 Java Deserialization Gadget Chains

Let us consider an insecure deserialization entry point, where a server deserializes incoming socket data into instances of *MyObj* (see Listing 1). It appears as if an attacker cannot inject arbitrary data into the *ObjectInputStream* without violating the cast to *MyObj* at line 3.

```
1  ObjectInputStream ois = new ObjectInputStream(socket.getInputStream());
2  MyObj obj = (MyObj) ois.readObject();
```

Listing 1. Insecure deserialization entry point.

Yet, upon deserialization, magic methods (e.g., *readObject()*) within the object to be deserialized get called. Any serializable object may override these methods to ensure compatibility between different Java versions [27]. Most importantly, these methods are called during deserialization, and thus, before the object gets cast to its intended object type. This becomes a security implication when it is possible to construct an object such that, using the initial *readObject()* and subsequent method calls, a security-sensitive method gets reached. The concept of chaining multiple method invocations (i.e., **gadgets**) to reach a security-sensitive method is called a **gadget chain**. Listing 2 shows an artificial gadget chain leading to remote code execution. The deserialization method *readObject()* at line 3 calls the method *LinkGadget.run()* at line 5 and, from there, continues to call *SinkGadget.run()* at line 11, ultimately resulting in the invocation of an arbitrary system command at line 15. Looking back at the insecure deserialization entry point in Listing 1, it is exploitable if a gadget chain (e.g., Listing 2) can be constructed from arbitrary classes loaded by the JVM.

```
1  class TriggerGadget implements Serializable {
2      LinkGadget linkGadget;
3      private void readObject(ObjectInputStream s) {
4          s.defaultReadObject();
5          linkGadget.run();
6  }}
7  class LinkGadget implements Serializable {
8      SinkGadget sinkGadget;
9      String command;
10     public void run() {
11         sinkGadget.run(command);
12 }}
13 class SinkGadget implements Serializable {
14     public void run(command) {
15         Runtime.getRuntime().exec(command);
16 }}
```

Listing 2. Synthetic example of a gadget chain.

Before continuing with a more complex example, we define the main terms to describe a gadget chain. We use the same terminology as in [52], where a **trigger gadget** refers to the first gadget within

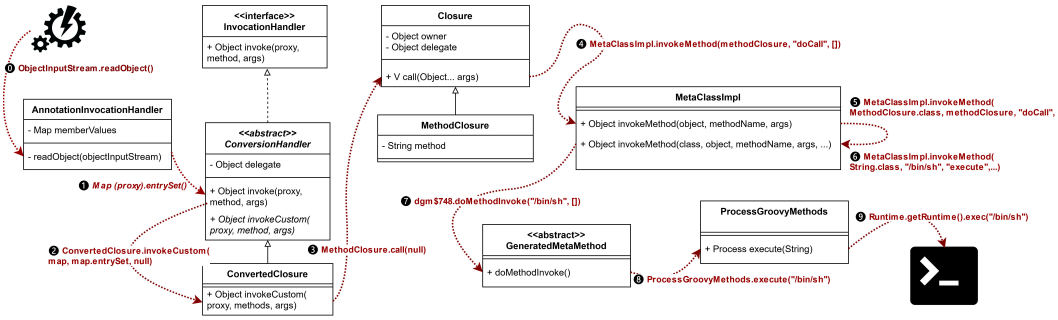


Fig. 1. Groovy1 gadget chain from the *Ysoserial* repository [24].

a gadget chain. A sequence of **link gadgets** connects a **trigger gadget** to a security-sensitive method, i.e., a **sink gadget**. Furthermore, we refer to common link gadgets invoked through *readObject()* as **trampoline methods**. For example, the trampoline methods *Object.hashCode()* and *Object.toString()* are accessible through trigger gadgets in the *Java Class Library* and are known to be frequently used at the beginning of gadget chains [44]. This concept is relevant for designing a benchmark since, to increase efficiency, algorithms may skip looking for trigger gadgets and instead start gadget chains from trampoline methods.

The *Ysoserial* gadget chain *Groovy1* [24] (see Figure 1) incorporates some of the challenges of gadget chain detection. The serialized object comprises an *AnnotationInvocationHandler* containing a proxied *Map* object, whose method calls will be handled through a *ConvertedClosure*. We omit the implementation relationship between the *InvocationHandler* and the *AnnotationInvocationHandler* in the figure to highlight that the purpose of this object is not to be used as a proxy but as a trigger gadget. In concrete terms, *readObject()* contains a call to *Map.entrySet()* (1), which is proxied by a *ConvertedClosure*. The detection of gadget chains using **dynamic proxies** is known to be one of the main limiting factors of gadget chain detecting algorithms [15, 52]. The *Groovy1* gadget chain exposes another difficult intermingling of Java programming concepts. In (3), *Java's* **runtime polymorphism** (inheritance) enables an object of type *MethodClosure* to be substituted for a *Closure* object. Furthermore, using precisely this object is crucial since, later, within the method call (5), a critical branch is only reached by determining the *owner* class being of the type *MethodClosure* via **reflection**. Finally, in (7), the *Groovy* framework calls a default *Groovy* method, which wraps *Java's Runtime.exec()*. This step is especially complex because finding the desired default *Groovy* method, referring to *ProcessGroovyMethods.execute()* (8) requires a string-based lookup within *Groovy's MetaClassRegistry*. The *MetaClassRegistry* is populated during runtime according to the external *META-INF/dgminfo* file[2][21]. A gadget chain detecting algorithm would have to reason about the contents of this file and the way it is interpreted to fully resolve the *Groovy1* gadget chain.

Indeed, the only way this gadget chain is currently discovered is by defining *MetaClass .invokeMethod()* as a sink method and thus assuming the *execute()* default *Groovy* method is registered [27]. Dissecting a single gadget chain (*Groovy1*) already sheds some light on the difficulties gadget chain detecting algorithms need to overcome. In the following section, we analyze patterns imposing such difficulties in detail so we can incorporate them into our benchmark.

[2]For details, we refer to *org.codehaus.groovy.runtime.metaclass.MetaClassRegistryImpl*, line 183 and *org.codehaus.groovy.reflection.GeneratedMetaMethod*, line 186-192 in the *Groovy* repository [21], tag *GROOVY_2_3_9*.

2.2 Challenges in Detecting Gadget Chains

The task of detecting gadget chains in Java applications is a specific use case for program analysis. Concretely, the path from trigger to sink gadgets can be abstracted to the overarching problems of execution and taint flow from an entry point (e.g., *readObject()*) to a sink method. As such, common challenges in program analysis also apply to uncovering gadget chains. These challenges are further enhanced by the inherently large search space originating from deserialization methods [48, 49, 52]. Together with *Java's* dynamic features as an object-oriented language, this imposes the following difficulties listed below. These challenges are derived from tool limitations and literature reviews [32–34].

(1) **Java Runtime Polymorphism** makes it difficult to reason about actual variable types from static analysis. Lacking any indication of the objects within a serialized data stream presents *Class Hierarchy Analysis* as the only viable option for determining potential runtime types. This is known to result in a path explosion problem [12, 14, 52].
(2) In alignment with large search space, there is no theoretical limit to **gadget chain length**. Algorithms may either stop at a certain depth [9, 12, 52] and/or timeout [9].
(3) The same argument applies to the **total number of gadget chains** existing in a given Java application. Specifically, it is known that some gadget chain detection algorithms (e.g., *Gadget Inspector* [27]) will not consider a gadget used in one gadget chain to potentially be reusable in another gadget chain [58]. Naturally, this performance-motivated cut leads to false negatives.
(4) The exploitation of gadget chains requires **complex payload generation** of nested and parallel objects [14]. Taint analysis fails to accommodate these intricacies and thus produces many false positives. This has sparked the notion of guided fuzzing to replicate the object structure and probe for actual exploitation to drastically reduce the number of false positives [11, 12, 14, 52]. However, even the most recent publications report not being fully able to resolve all constraints in payload object generation, thus necessitating manual verification to remove false positives [14, 52].
(5) **Java Lambda Expressions** are implemented using the special *invokedynamic* bytecode instruction to dynamically bootstrap the underlying function call site [19]. This feature is not soundly modeled by all static analysis frameworks, as shown in [53]. Since gadget chain detection algorithms usually rely on static analysis frameworks such as Doop [44] or Soot [9, 52], considering Java Lambda expressions is imperative.
(6) **Runtime Exceptions**, as indicated by [13], are barely touched upon in static analysis.
(7) **Java native method invocation (JNI)** adds another challenge to the detection of gadget chains [33]. Research regarding precise inter-language data flow is ongoing [34, 57], but has yet to be incorporated by static analysis frameworks [53]. These will instead fall back (if at all) to modeling the behavior of the call according to its expected native implementation.
(8) The **Reflection API** is both a frequently used feature of the Java platform and also particularly pernicious towards static analysis [23, 32, 44, 49]. In fact, gadget chain detection algorithms often resort to flagging certain reflection methods, such as *Method.invoke()*, as sink methods without further consideration [9]. Similarly, dynamic proxies (*InvocationHandler.invoke()*) are assumed to be blanket trampoline methods [27].
(9) Finally, in order to holistically solve gadget chain detection, the entirety of **serialization entry points and sink methods** needs to be considered. Alternative serialization libraries offer further entry point methods, which are prone to similar exploitation techniques as default Java serialization [2]. Simultaneously, generating a comprehensive list of all sink methods for a given Java application remains an unsolved challenge [52]. However, static lists of known sink methods contain at least 25 entries [12]. Again, this highlights the vast search space for gadget chains.

With the exception of (9), we aim to represent all these challenges in our benchmark. We exclude (9) to keep the benchmark sink-agnostic. In fact, it would be detrimental for a benchmark to expose the currently known set of sink methods since it would a) deprecate itself with new sink gadgets being discovered and b) skew results towards algorithms with exhaustive sink method lists rather than those most mature in detecting actual gadget chains. The challenge of detecting Java sink methods is tackled separately, e.g., in [28, 45].

3 Design and Implementation

For our benchmark chains, we conceptualize a single entry- and exit-point model (see Figure 2). Since gadget chain detection algorithms might only look for common trampoline methods instead of all overridden *readObject()* methods [44, 52], we use a single trigger gadget, calling the trampoline method *hashCode()* on all gadget chains. Notice that all trampoline gadgets are a subclass of *GleipnerObject*. Conversely, this forces gadget chain detection tools that start their search from *readObject()* to stay within the boundaries of the benchmark.

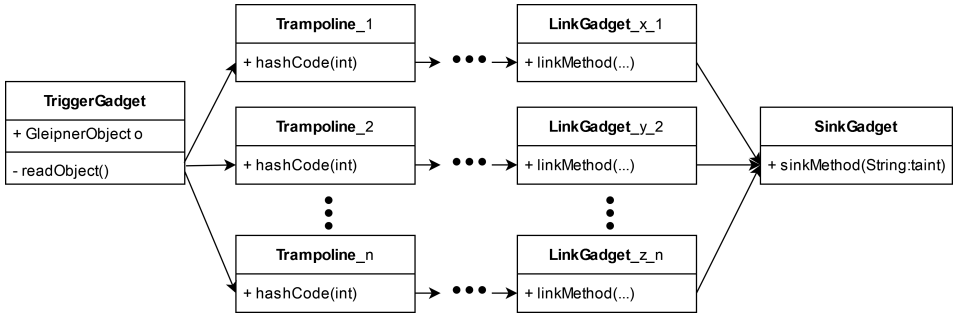


Fig. 2. Simplified architecture of *Gleipner*.

However, only considering trampoline methods (e.g., *hashCode()*) is a limitation, which should be revealed by a benchmark. Therefore, we deviate from the architecture in Figure 2, adding four gadget chains with the deserialization methods *readObject()*, *readResolve()*, *readObjectNoData()*, and *readExternal()* directly connected to the sink gadget. We limit ourselves to the Java native deserialization methods since it is not our objective to probe for exhaustive entry point lists. It is merely a cheap addition to test whether algorithms employ the trampoline method approximation. We also add two test cases to probe the algorithm's sensitivity to essential gadget chain keywords, i.e., implementing the *Serializable* interface and correctly interpreting non-serializable (*transient*) properties.

In Sections 3.1 to 3.3, we elaborate on how we emulate the individual challenges in gadget chain detection. Additionally, we replicate *Ysoserial* gadget chains in Section 3.4 to include real-world-related, compound gadget chains, incorporating these challenges.

3.1 Emulating Large Search Space

The main contributing factors to the large search space are determined by (1) depth, (2) inheritance complexity, and (3) the number of paths to a gadget chain. As discussed in Section 2, the maximum depth and number of paths may be limited by the gadget chain detection approach, while polymorphism most likely impacts performance. We benchmark the individual tools' sensitivity to all three traits by generating chain structures, as shown in Figure 3.

The simplicity of these structures supports the automation of constructing these chains from class templates. Intuitively, the complexity is defined by the number of link gadgets between trigger

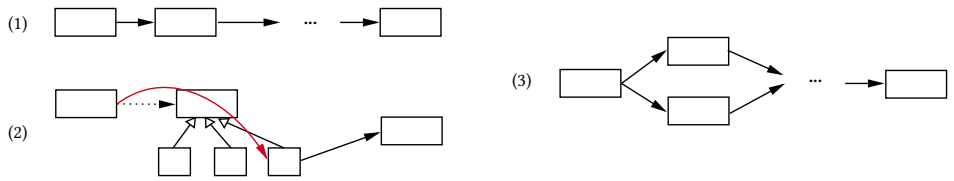


Fig. 3. Synthetic chains for testing (1) depth, (2) polymorphism and (3) multiple paths.

and sink gadgets in (1) and the number of branching link gadget invocations in (3). In (2), we both control the number of children per inheritance layer and the inheritance depth. The sink gadget is thereby always located on the bottom layer.

3.2 Emulating Complex Payload Generation

To replicate deeply nested and parallel object structures, we create three helper classes (see Listing 3). This trivial structure allows payload objects to grow both horizontally (by adding more items to the array *nArray*) and vertically (by populating the arrays of the contained nested items). Then, the *NestedString* and *NestedSink* can be placed anywhere within this structure to control the tainted data and the sink method call, respectively.

```
1  public class Nested implements Serializable {
2      Nested[] nArray;
3      public Nested(int size) {
4          this.nArray = new Nested[size];}
5      public Nested get(int i) {
6          return this.nArray[i];}
7      public void set(int i, Nested n) {
8          this.nArray[i] = n;}
9      public String invoke(String arg) {return null;}
10 }
```

```
11 public class NestedString extends Nested {
12     String value;
13     public String invoke(String arg) {return this.value;}
14 }
15 public class NestedSink extends Nested {
16     SinkGadget sink;
17     public String invoke(String arg) {
18         if (arg != null) sink.sinkMethod(arg);
19         return null;
20 }}
```

Listing 3. Nested payload class structure.

We illustrate how to apply this concept to construct synthetic gadget chains in Figure 4. Given a reachable gadget (e.g., a trampoline method as in Listing 4), the taint propagation to a *NestedSink* at line 7 can only be satisfied by using an object structure as in Figure 4.

```
1  public TrampolineGadget extends GleipnerObject {
2      Nested nested;
3      public int hashCode() {
4          Nested sink = nested.get(0).get(0);
5          Nested cmd = nested.get(0).get(1);
```

```
6          if (cmd.equals(nested.get(1)) {
7              sink.invoke(cmd.invoke(null));
8          }
9          return 0;
10 }}
```

Listing 4. Nested payload example.

To effectively benchmark, we also include false positive trampoline gadgets, where it is either impossible to taint or reach the sink method. For instance, we modified the conditional statement at line 6, Listing 4, to *cmd.equals(sink)*. Thus, although both *cmd* and *sink* can be tainted, this constraint makes it impossible to have the required combination of a *NestedSink* and a *NestedString* available at line 7. We predict that purely static gadget chain detection algorithms will find all synthetic gadget chains, including false positives, while hybrid approaches may fail to find all true positives due to being incapable of generating an applicable payload.

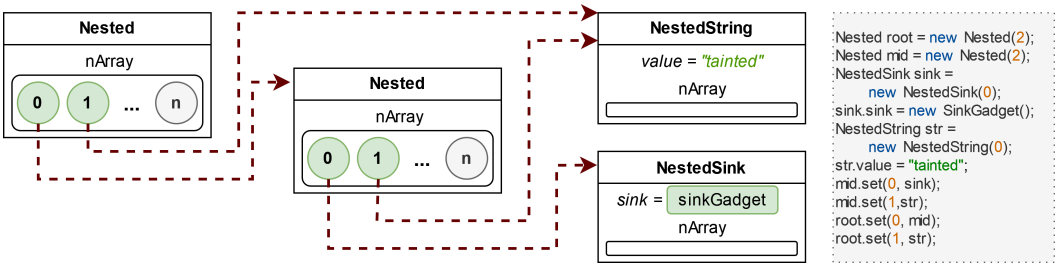


Fig. 4. Nested object payload.

3.3 Emulating Dynamic Java Features

We include synthetic gadget chains using JNI, runtime exceptions, the *invokedynamic* bytecode instruction, and the *Reflection API*. Thereby, we mainly focus on comprehensively representing the latter due to its frequent usage[3] and the known difficulties it causes in program analysis [23, 32, 53]. Indeed, it was shown in [32] that static analysis tools come with their individual sets of limitations, such as ignoring exceptions thrown by the usage of reflection or assuming the absence of dynamic proxies. We take this as a rationale to thoroughly incorporate the Reflection API into our benchmark.

To achieve this, we consider the 17 categories of Java reflection described by *Landman et al.* [32]. Of these categories, four are capable of invoking link gadgets (see Table 1). *Load Class* not only represents class initialization (*clinit*), calling static blocks within the class being initialized, but also loading a class into the JVM with a custom classloader. *Construct Object* and *Invoke Method* are the reflective equivalent of calling a class's constructor or any other method. Finally, *Dynamic Proxies* are probably the most difficult to reason about in static analysis since whenever an interface method is called, it may not be handled as defined by the implementing class but instead be delegated to an invocation handler.

Table 1. Reflection methods linking to arbitrary call site.

Category	Reflection Methods
Load Class	*Class.forName(), ClassLoader.loadClass()*
Construct Object	*Constructor.newInstance(), Class.newInstance()*
Invoke Method	*Method.invoke()*
Dynamic Proxies	<any interface method>

Gadget chain detection algorithms often assume methods in these four categories to causally be sink or trigger gadgets [9, 27]. This assumption is most evident with *Method.invoke()*, being able to call arbitrary methods (including sink methods). However, the problem with this approach is that it makes no effort to model the *Java* reflection API. A sound model would have to traverse all possible call sites reachable from *Method.invoke()*. For example, in Listing 5, given that the attacker can taint all fields (lines 2,3), calling *method.invoke()* at line 5 would only have a security implication if the class *Dummy* contains any gadgets, which can be further utilized in a gadget chain. For this very reason, we include at least one true and false positive gadget chain for each of these reflection-based method calls. Under these conditions, a gadget chain detection algorithm can only be considered

[3]An estimate of 78% of projects use at least one feature of the Java Reflection API [32].

sound and precise towards Java reflection if it both reports the true positives and none of the false positives.

```
class LinkGadget implements Serializable {
    Dummy d;
    Class<?>[] argTypes;   Object[] args;    String methodName;
    public void linkMethod() {
        Method m = d.getDeclaredMethod(methodName, argTypes);
        m.invoke(d, args);
}}
```

Listing 5. Example link gadget using reflection.

While the remaining categories provide no capability of connecting to link gadgets, they still need to be correctly interpreted to ensure accurate control flow and taint analysis. According to [32], static analysis tools are especially limited by inaccurate (or no) modeling of:

(1) **Exceptions** due to reflection (e.g. *ClassCastException*)
(2) **Collections of *MetaObjects***[4] and filtering thereof within control flow blocks
(3) any use of **Dynamic Proxies** (already covered by Table 1)

We benchmark (1) by creating both one false and one true positive test case for all exception subclasses of *ReflectiveOperationException* [40] and the runtime exception *ClassCastException*. To cover (2), we develop test cases related to usage patterns in real-world applications. That is, traversing *MetaObjects* signatures and reading/updating fields or modifiers [32].

In addition to the Java reflection API, we supplement our benchmark with gadget chains relying on other JNI methods. We consider both the native call to *Thread.start()* and one example backed by a custom native library. This setup is similar to the micro benchmark in [53]. Regarding Java Lambda expressions, we add one true and one false positive, respectively. Also, we select 7 classes inheriting from *RuntimeException* with predictable (and thus reproducible) trigger conditions (e.g., *NullPointerException* or *IndexOutOfBoundsException*).

3.4 Implementation of Ysoserial Chains

The aim of implementing known gadget chains from the *Ysoserial* repository in our benchmark is to provide a set of tests closely related to real-world gadget chains. In contrast to all other tests, the *Ysoserial* chains are complex and composed of multiple analysis challenges. Hence, a gadget chain detection algorithm's capacity to find these chains is an attempt to approximate its overall utility outside a lab environment.

As has been discussed in Section 1, appending the dependencies containing *Ysoserial* gadget chains adds little value to a benchmark. This is, in general, due to the missing knowledge of the ground truth within the dependencies. Instead, we manually analyze and re-implement the gadgets in our benchmark. Thereby, we abstract parts of the gadgets' code where it interacts with APIs within the dependency, which are generally irrelevant to the chain's execution path. Furthermore, we replace the sink method and parameters propagating towards it with *Gleipner's SinkGadget.sinkMethod()*. For *Ysoserial* gadget chains relying on trampoline gadgets, we ensure they connect to the *Gleipner TriggerGadget.readObject()*. However, for gadget chains that do not rely on trampoline methods (e.g., *C3P0*), we keep their individual *readObject()* entry point. Doing so highlights the shortcomings of gadget chain detection algorithms which only consider trampoline methods instead of all overridden serialization methods. Using the *C3P0* gadget chains as an example, we explain our process of abstracting a *Ysoserial* gadget chain. For the implementation of the remaining *Ysoserial* chains, we refer to our repository link in Section 9.

[4]Objects of type *Class*, *Method*, *Field* or *Construtctor*.

Table 2. C3P0 gadget chain.

com.mchange.v2.c3p0.impl.**PoolBackedDataSourceBase->readObject**
↪ com.mchange.v2.naming.**ReferenceIndirector$ReferenceSerialized->getObject**
↪ com.mchange.v2.naming.**ReferenceableUtils->referenceToObject**
↪ java.lang.**Class->forName**

The *C3P0* gadget chain is composed of 4 gadgets (see Table 2), with the sink method *Class.forName()* loading an arbitrary remote class using a tainted *URLClassloader*. To re-implement the gadget chain, we first retrieve and decompile the dependency JAR files in the exploitable version noted by *Ysoserial*. Specifically, we download *c3p0 0.9.5.2* and *mchange-commons 0.2.11* from the *Central Maven Repository* [39]. Then, beginning with the trigger gadget, we keep all method calls in *readObject()* up to the point where the link gadget is invoked and discard the remaining code (see Listing 6). We also strip *PoolBackedDataSource* of all properties and methods not being utilized by the gadget chain.

```
1   private void readObject(ObjectInputStream ois) {
2       short version = ois.readShort();
3       switch (version) {
4           case 1:
5               Object o = ois.readObject();
6               if (o instanceof IndirectlySerialized) {
7                   o = ((IndirectlySerialized)o).getObject();
8               } // removed logic
9               return;
10      default:
11          throw new IOException("[...]" + version);
12  }}
```

Listing 6. *PoolBackedDataSourceBase.readObject()*.

In *ReferenceSerialized.getObject()*, we cut ten lines of code before reaching the next link gadget. The rationale is that these method calls have no effect on the actual tainted parameter (of type *Reference*) used in the sink. One could argue that this deviation functionally deprives the original chain of potentially crashing during the removed method invocations, thus preventing the gadget chain from executing. However, by the same token, these method invocations could lead to further unplanned gadget chain findings in our benchmark. For this reason, we opt for the trade-off of keeping the gadget chains only closely related to the original, yet at the same time predictable. Regarding *ReferenceableUtils.referenceObject()*, we replace the known sink method *Class.forName()* with an invocation of the *Gleipner* sink method. In a similar process to the one described for *C3P0*, we implement **12** true positive *Ysoserial* gadget chains in our benchmark. We chose these chains specifically for the challenges exposed (see repository description, Section 9). Furthermore, we append one real-world, false positive gadget chain found when analyzing the *Aspectj* dependency with *GadgetInspector*.

4 Experimentation and Evaluation

4.1 Benchmarked Tools

To our knowledge, and at the time of writing, there are 12 gadget chain detection tools (see Table 3). Tools marked as hybrid include a fuzzing module to verify gadget chains found during static analysis. For four (*33%*) tools, we could not find an accompanying artifact. In addition, *GCMiner* and *JDD* are provided as incomplete artifacts. *GCMiner* is missing substantial portions of its source code, making it entirely unusable[5], whereas the *JDD* artifact is provided without the fuzzer described in the publication [14]. We contacted the authors of *HawkGadget*, *GCGM*, *GCMiner*, *ODDFuzz*, *RevGadget*, and *JDD*. *ODDFuzz* and *GCMiner* will remain closed-source until reviewed by the company backing

[5]As supported by an open issue (https://github.com/GCMiner/GCMiner/issues/1)

Table 3. Gadget chain detecting tools.

Tool	*Serianalyzer* [3]	*GadgetInspector* [27]	*SerHybrid* [44]	*HawkGadget* [58]	*GCGM* [31]	*JChainz* [9]	*Tabby* [15]	*GCMiner* [12]	*ODDFuzz* [11]	*Crystallizer* [52]	*RevGadget* [36]	*JDD* [14]
Year	'17	'18	'20	'22	'22	'22	'23	'23	'23	'23	'24	'24
Availability	✓	✓	✓			✓	✓	(✓)		✓		(✓)
Hybrid			✓					✓	✓	✓		✓

the project. Further, the fuzzing module in *JDD* is said to undergo quality assurance before being published. However, we can still benchmark the static analysis module of the tool. We received no answer from the authors of *HawkGadget*, *GCGM* and *RevGadget*. Consequently, we evaluate **seven** tools (including *JDD*) against our benchmark. This number is still representative of the state-of-the-art since we cover recent [14, 15, 52], purely static [3, 9, 15, 27], and hybrid approaches [44, 52]. We now briefly describe these tools and explain how they were modified to specifically target our benchmark. For some tools, we had to adjust hard-coded paths, dependencies and JVM versions. Since these modifications are unrelated to the benchmark itself, we refer to our repository (see Section 9) for further details on that matter.

GadgetInspector [27] and *Serianalyzer* [3] are two tools similar in concept. They both rely purely on static analysis techniques and are implemented with the Java ASM bytecode analysis framework [8]. In regard to the analysis, they perform basic taint analysis to detect whether an attacker-controlled value can propagate to a sink method. Sink methods are determined by method name comparison in the *GadgetChainDiscovery.java* and *SerianalyzerConfig.java* source code files, respectively. We modify these files to only search for *Gleipner's* sink method, *SinkGadget.sinkMethod()*. Note that both tools are insensitive to conditional statement satisfiability and are, therefore, prone to producing many false positives.

SerHybrid [44] is the first tool to attempt dynamic gadget chain validation. It performs both static analysis to determine potential gadget chains and also fuzzes the inspected library with *Randoop*. However, we were unable to determine a coupling between the static and dynamic analysis of the tool within the tool's source code. I.e., the output of the static analysis is not used to direct the fuzzing. As such, we found that *Randoop* is barely capable of finding any gadget chains, and we discard those results in favor of the findings from the static analysis. The static analysis is performed using a specific version of *Doop*[6], running custom *Datalog* scripts. We launch *Doop* with the *–Xextra-logic* flag to include these rules. We also modify the *sinks.txt* file to specifically search for the *Gleipner* sink method.

JChainz [9] is another purely static analysis tool for gadget chain detection. However, in contrast to *GadgetInspector* and *Serianalyzer*, it performs an additional verification step to cull false positives. This step includes custom taint propagation logic, which focuses on data type constraints of attacker-controlled parameters being passed through the chains. No configuration is needed since the sink method is passed as a command-line argument to the tool. However, we observe a flaw in the logic of *JChainz*, where the data type analysis fails to recognize a gadget being able to hold gadgets of a subtype during invocation. For example, in Listing 7, the *Finder* component of *JChainz* successfully detects the gadget chain from *TriggerGadget.readObject()* to *LinkGadget.hashCode()*. Yet, during the *Analyzer* routine, the data type dependency analysis culls the connection between *LinkGadget* and *TriggerGadget* due to an alleged type inconsistency between *GleipnerObject* and

[6] https://bitbucket.org/yanniss/doop/commits/ba731a63c90e94f9e94afca39ff15c5082bf868d, note that this version requires an older version (v1.7.0) of *Soufflé* to be installed.

LinkGadget. The majority of gadget chains in the *Gleipner* benchmark are set up this way (see Figure 2 in Section 3). Thus, in order to fairly benchmark *JChainz*, we compile a modified version of the benchmark, where the *hashCode()* methods of *GleipnerObject* trampolines are replaced with a call to *readObject()*. This ensures that the tool correctly finds the entry point, and going from there, the tool's performance in finding the actual gadget chain can be examined.

```
1  class GleipnerObject implements Serializable { }
2  class TriggerGadget implements Serializable {
3      public GleipnerObject o;
4      private void readObject(ObjectInputStream ois) {
5          o.hashCode();
6  }}
7
8  class LinkGadget extends GleipnerObject {
9      public int hashCode() {
10         SinkGadget.sinkMethod(taint);
11 }}
```

Listing 7. False negative in JChainz.

Tabby is a framework to extract a *deserialization-aware call graph* (DA-CG) from Java applications and export it into a *Neo4J* database for later reusability [15]. The idea of the DA-CG is, in general, to reduce the problem of finding gadget chains to a graph reachability problem with a small set of relationship edges. Therefore, the majority of static analysis, and in particular graph pruning, is performed during graph construction. We configure *Tabby* to label the *Gleipner* sink method as *IS_SINK* in the *sinks.json* file. Furthermore, to import the DA-CG generated by *Tabby* to the Neo4j database, we clone and build the separate tool *tabby-vul-finder*[7]. To perform the path search on the DA-CG, we rely on an *Apoc* query statement, similar to what has been done in [12] (see Listing 8). Thereby, we control the maximum length of the gadget chain with the fourth parameter in the call to *allSimplePaths()*. Note that the maintainers of *Tabby* also provide a custom gadget chain finding routine[8] for *Neo4J*. However, when querying the generated DA-CG, we were unable to find any gadget chains at all and therefore resorted to the *Apoc* routine.

```
MATCH (source:Method {Name: "readObject"})
MATCH (sink:Method {IS_SINK: true})
CALL apoc.algo.allSimplePaths(sink, source, "<", 30) YIELD path
RETURN path
```

Listing 8. Cypher statement Tabby.

The process behind *Crystallizer* [52] is fourfold in an alternating static-dynamic sequence. First, the tool creates an initial gadget graph based on a pruned class hierarchy analysis. The graph is then used to fuzz potential reachable sink methods based on their ability to instantiate arbitrary classes. In the third step, with the initial gadget graph, *Crystallizer* executes a path search from entry points to the previously identified sinks. Finally, the gadgets within found paths are used to guide object formation for another fuzzer. The main difficulty imposed on our benchmark by this setup is the dynamic sink identification. While with all other tools, the sink method can simply be appended to the static rules, *Crystallizer* determines the list of sinks used for gadget chain detection during runtime. Our modifications of the tool aim at short-circuiting the dynamic sink identification in favor of replacing it with the *Gleipner* sink method. To do so, we comment out the dynamic sink search in the main *run_campaigns.sh* script. Then, because the static filter rules expect a certain output from the dynamic sink identification, we create a mock file containing only the *Gleipnir* sink method signature. Furthermore, the static filter rules will attempt to filter out dynamic sink findings. Therefore, we remove the two checks in *SinkAnalysis.java* (line 90 and line 109). We also

[7] https://github.com/wh1t3p1g/tabby-vul-finder
[8] https://github.com/wh1t3p1g/tabby-path-finder

adjust the *LibSpecificRules.java* file to include our trampoline method, as indicated by the authors in their repository's description[9].

Crystallizer's guided fuzzer instantiates classes via their available constructor only (see *src/dynamic/Meta.java*, line 865 ff). This is a limitation to be aware of since, consequently, the fuzzer is incapable of modifying object properties set independently of the constructor, which may be crucial for the gadget chain to function. Keeping this limitation in mind, we add a parameterized constructor to every non-ysoserial gadget chain in the benchmark.

JDD [14] introduces two new concepts for the detection of gadget chains. First, the static analysis proportion of the tool constructs higher-level gadget fragments, which summarize all method calls contained within one object (i.e., are not dynamically dispatched). These fragments are then chained together in reverse order from sink to source fragments. Second, to guide the fuzzer, JDD performs a path- and context-sensitive extraction of fields within gadget fragments to create a skeleton of an injection object including dataflow-dependent constraints. The constraints are used within the mutation strategy of the JQF-based [42] fuzzer. To run *JDD* on our benchmark, we modified one of the sink check rules[10] to search for the *Gleipner* sink method and then reference it in the *JDD* configuration file.

4.2 Setup

Before building, we generate 50 gadget chains for the emulation of large search space (see Section 3.1)[11]. These are 10 multipath chains, 20 depth chains with a depth of 4 up to 23, and 20 polymorphism chains with up to 10 inheritance layers and varying amounts of children per layer. The largest test case in the latter category consists of 2 047 ($2^{11} - 1$) classes for the inheritance structure. We also verify the true positive gadget chains in the benchmark with unit tests. Specifically, we serialize and deserialize a payload object, which triggers the *SinkGadget.sinkMethod()*. The sink gadget itself sets a flag in a static *Controller* class as proof of its invocation. Thus, we can assert that this flag was set. Thereafter, we build the *Gleipner* gadget chains into separate JAR files for each category being tested. Note that if a tool fails to find all chains in the multipath category, we need to split up the payload generation gadget chains into separate *JAR* files. This is because the *Nested* class is being reused as a gadget, and thus a tool may not find gadget chains due to insensitivity to gadget reuse rather than the supposedly tested payload generation.

We run the benchmarked tools over this collection of JAR files. If required, we set timeout parameters according to the authors' recommendations [12]. In the case of *Crystallizer*, we first run the fuzzer with a timeout of 60 seconds, and only if the fuzzer could not confirm all chains found during the static analysis we set it to 24 hours. The experiment is run on the Debian Linux 5.10.197 OS, with a 64-core AMD EPYC 7713P processor (2.00 GHz) and 995 GB of RAM. The *Gleipner* benchmark and evaluator module are written in Java. Apart from that, we created an instrumentation script to automate running the benchmark on the seven tools and two scripts for synthetic gadget chain generation in Python.

4.3 Results

The results are presented in Table 4. For readability, true positive (TP) cells are shaded red if not all true gadget chains were found. Conversely, false positive (FP) cells are highlighted in red if any false gadget chains were found. This metric is also used to determine the number of satisfied categories (final column), which indicates in how many categories a tool is sound and precise in

[9] https://github.com/HexHive/Crystallizer/tree/main
[10] https://github.com/fdu-sec/JDD/blob/main/src/jdd/rules/sinks/ExecCheckRule.java
[11] The remaining gadget chains require no further setup.
[12] *24* hours for *Crystallizer* and *2* hours for *JChainz*.

Table 4. Results by tool and category: true positives (*TP*), false positives (*FP*) and execution time (t_s). A category is considered satisfied by a tool if all true positives and no false postivies were detected.

		Depth Tests	Polymorphism Tests	Multipath Tests	Payload Construction	Serialization API	Serialization Keyword	Reflection API							Runtime Exceptions	Other JNI Methods	Invoke Dynamic (J)	Ysoserial	Total Findings	Satisfied Categories
								Method.invoke()	*Class.initialization*	*Classloading*	*Constructor*	*Dynamic Proxies*	*Exceptions*	*Meta Objects*						
(1) Serianalyzer	TP	20/20	20/20	1/10	9/9	3/4	2/2	1/5	0/2	1/2	1/3	1/3	7/7	11/11	7/7	1/2	1/1	6/12	92/120	**2**
	FP	0/0	0/0	0/0	6/6	0/0	1/2	1/5	0/2	1/1	1/3	1/2	7/7	10/10	7/7	0/0	1/1	0/1	35/47	
	t_s	0.2s	3.8s	0.2s	0.2s	0.2s	0.1s	0.2s	0.1s	0.2s	0.2s	0.2s	0.2s	0.2s	0.2s	0.2s	0.2s	1.6s		
(2) GadgetInspector	TP	20/20	20/20	1/10	2/9	1/4	1/2	1/5	0/2	1/2	1/3	3/3	7/7	11/11	7/7	0/2	0/1	5/12	81/120	**2**
	FP	0/0	0/0	0/0	0/6	0/0	0/2	1/5	0/2	1/1	1/3	2/2	7/7	9/10	7/7	0/0	0/1	1/1	28/47	
	t_s	31.7s	30.7s	30.5s	31.1s	30.5s	29.4s	28.8s	29.2s	27.9s	29.4s	29.3s	31.4s	29.8s	29.3s	29.6s	30.8s	368.7s		
(3) SerHybrid	TP	13/20	20/20	10/10	1/9	0/4	2/2	0/5	0/2	1/2	1/3	0/3	7/7	8/11	7/7	1/2	0/1	2/12	73/120	**2**
	FP	0/0	0/0	0/0	1/6	0/0	1/2	0/5	0/2	1/1	1/3	0/2	7/7	8/10	6/7	0/0	0/1	1/1	26/47	
	t_s	201.1s	556.5s	185.6s	184.0s	180.2s	123.6s	111.0s	108.7s	112.2s	112.3s	118.8s	123.8s	146.5s	112.0s	184.1s	119.4s	2420.3s		
(4) JChainz	TP	20/20	0/20	10/10	9/9	1/4	1/2	1/5	0/2	1/2	1/3	1/3	7/7	10/11	7/7	0/2	0/1	0/12	69/120	**2**
	FP	0/0	0/0	0/0	6/6	0/0	0/2	1/5	0/2	1/1	1/3	1/2	7/7	10/10	6/7	0/0	0/1	0/1	33/47	
	t_s	671.3s	1086.1s	231.3s	8087.9s	89.5s	125.8s	438.6s	49.4s	187.4s	281.4s	145.4s	36440.6s	1448.9s	350.5s	37.3s	36.9s	60334.2s		
(5) Tabby	TP	20/20	20/20	10/10	9/9	4/4	2/2	1/5	0/2	1/2	1/3	1/3	7/7	10/11	7/7	0/2	1/1	7/12	101/120	**4**
	FP	0/0	0/0	0/0	6/6	0/0	1/2	1/5	0/2	1/1	1/3	1/2	7/7	10/10	6/7	0/0	1/1	0/1	35/47	
	t_s	26.2s	33.8s	25.5s	24.8s	23.2s	28.5s	24.5s	19.4s	44.9s	18.7s	19.0s	28.1s	42.7s	24.9s	27.8s	39.5s	286.5s		
(6) Crystallizer	TP	3/20	0/20	10/10	0/10	0/4	2/2	0/5	0/2	1/2	1/3	1/3	6/7	9/11	3/7	0/2	0/1	4/12	41/120	**2**
	FP	0/0	0/2	0/0	0/6	0/0	0/2	0/5	0/2	0/1	0/3	1/2	0/7	0/10	0/7	0/0	0/1	0/1	1/47	
	t_s	106.2s	86497.8s	120.5s	86461.9s	105.1s	104.8s	86457.3s	115.7s	104.6s	101.3s	86455.6s	86458.2s	73305.4s	86456.3s	104.5s	107.0s	87862.7s		
(7) JDD	TP	5/20	0/20	10/10	0/10	3/4	2/2	0/5	0/2	1/2	1/3	0/3	7/7	8/11	7/7	0/2	0/1	7/12	51/120	**1**
	FP	0/0	0/0	0/0	0/6	0/0	1/2	0/5	0/2	1/1	1/3	0/2	7/7	8/10	6/7	0/0	0/1	0/1	24/47	
	t_s	311.4s	319.8s	287.8s	255.2s	261.1s	256.9s	261.6s	253.2s	308.6s	254.0s	255.3s	322.6s	324.3s	283.2s	265.5s	259.8s	3745.4s		

gadget chain detection. This notion deserves to be emphasized: there is little utility in counting the total findings over all categories because categories are represented by different quantities of gadget chains. Rather, meaning is derived from the number of challenges accurately covered by the individual tools. Also, note that even though the fuzzer for *JDD* was unavailable (see Section 4.1), one can at least make observations regarding the lack of true positives found in the benchmark.

Overall, the results show that gadget chain detection tools are both far from being mature, as well as newer tools being subject to substantial performance degradation. The latter finding is uncovered thanks to the granular setup of test cases in our benchmark. For instance, evaluating the most recent publications *Crystallizer* [52] and *JDD* [14] in the context of large search space

reveals that both tools unsoundly handle long gadget chains[13] (see column 3, *Depth Tests*) and deep inheritance structures (column 4). These challenges in gadget chain detection are, however, handled by the earliest tools: *Serianalyzer* [3] and *GadgetInspector* [27]. By stating this, we do not intend to finger-point at the authors of new publications. It is evident that trade-offs had to be made to improve the tools in other areas (e.g., reducing false positives). Previously the authors could not transparently evaluate these areas of performance degradation because there was no proper benchmark to work with.

Apart from the challenge-based test cases, *Gleipner* also attempts to re-implement *Ysoserial* gadget chains. The tools' performance on these is depicted in Table 5. Chains highlighted in gray are those for which *Method.invoke()* cannot be trivially replaced with Gleipner's *SinkGadget.sinkMethod()*. For the *CommonsCollections*, this is due to the unique setup of the notorious *Transformer* chain. The other three chains execute a specific *Method.invoke()* instruction, which can only target *getter* methods, i.e., the method is only partially taintable due to the *get* prefix. Consequently, for these four gadget chains, we reconfigure the tools to search for chains using *Method.invoke()* as a sink gadget. Doing so also results in some of the tools' outputs being filled with false positives, which highlights the importance of using a custom sink method for the benchmark.

Table 5. Performance on ysoserial-based chains.
✓- Synthetic gadget chain was detected by tool.
- Result on synthetic chain aligns with publication result on *Ysoserial*.
- Result on synthetic chain does not align with publication result on *Ysoserial*.
- Unclear (which gadget chain was found in the underlying dependency).

Gadget Chain	(1)	(2)	(3)	(4)	(5)	(6)	(7)	entrypoint
aspectjweaver						✓	✓	*hashCode()*
c3p0	✓	✓			✓		✓	*readObject()*
cc1		✓					✓	*(Proxy) invoke()*
clojure	✓	✓			✓		✓	*hashCode()*
groovy								*(Proxy) invoke()*
hibernate1	✓				✓			*hashCode()*
jrmplistener	✓				✓			*readObject()*
myfaces1	✓	✓	✓		✓	✓	✓	*hashCode()*
rome						✓	✓	*hashCode()*
spring1								*readObject()*
urldns	✓	✓	✓		✓			*hashCode()*
vaadin1					✓	✓	✓	*hashCode()*

We compared the gadget chain findings with the results reported in the tools' publications (if available). In Table 5, the ticks (✓) indicate whether the synthetic chain in our benchmark was found by the tool. Then, squares highlighted in green denote if the benchmark results align with the publications, and red if not. In some cases, it is not clear which *Ysoserial* gadget chain was found by a tool within a dependency (highlighted in yellow). Thus, we rerun *SerHybrid* (2), *Tabby* (5), *Crystallizer* (6), and *JDD* (7) against the actual dependencies. For *Tabby*, the sink

[13]We should mention that now, with the help of *Gleipner*, the authors of JDD have fixed their tool by removing a hard-coded depth limit in *SearchUtils.java* (see: https://github.com/fdu-sec/JDD/commit/9f6a1a9948d8e6d02fcc0eee2f77ebb07a8f325e). The new version finds all 20 depth chains.

gadgets are only reached through false positive gadgets in the *HashSet* class or not at all. Both *Crystallizer* and *SerHybrid* are unable to find the CommonsCollections1 gadget chain. We specifically reimplemented this chain to have the additional challenge of using a dynamic proxy as an entry point. I.e., CommonsCollections5 is identical to the former but uses a *toString()* trampoline instead of the proxy. Indeed, both tools only detect the latter. We first run *JDD* against the *c3p0* dependency in isolation, and then, not having found the *Ysoserial* gadget chain, notice that this gadget chain only exists in conjunction with gadgets from the *mchange-commons*[14] library. After combining the two dependencies into a single JAR and rerunning *JDD*, the gadget chain is detected. Indeed, it seems the authors of *JDD* made the same mistake and, therefore, did not report the finding in their results section. In a way, this occurrence underlines that *Ysoserial* is not a benchmark.

With these results, we conclude that it is feasible to synthetically replicate real-world gadget chains for the benchmark. One could have expected the tools to perform better on the simplified replicas in *Gleipner*. If anything[15], the converse is the case. This implies that if a tool finds the benchmark gadget chain, it would also detect it in the real dependency.

5 Discussion

As a basis for our discussion, we consider the study in [55] evaluating the prevalence of benchmarking flaws in security systems. Note that this study specifically analyzed the way publications on mitigation techniques benchmarked their approaches in terms of incurred performance overhead (e.g., execution time with and without the mitigation technique). This means not all of the 22 listed flaws in [55] are applicable to evaluation malpractices in gadget chain detection tools. As such, we reflect on how design decisions for *Gleipner* help to alleviate specifically those issues related to gadget chain detection. The identifiers (e.g., A1) refer to the same identifiers used in [55].

(1) **Selective benchmarking:** we split up challenges in gadget chain detection tools and Java program analysis as a whole into fine-grained test cases. Thus, by evaluating tools against these challenges in isolation it becomes visible where trade-offs were made to improve the tool in another area, which is crucial to communicating **A1 performance degradation**. Also, *Gleipner* is provided as a dedicated benchmark. While previously, tools arbitrarily chose real-world dependencies against which to run their tool, now it is difficult to justify why a tool should only be tested against parts of *Gleipner*. This alleviates both **A2 benchmark subsetting without proper justification** and **A3 Selective data sets that hide deficiencies**.

(2) **Using the wrong benchmarks,** specifically **C3 Same dataset for calibration and validation**. Given the small set of real-world dependencies that contain gadget chains as per *Ysoserial* encourages overfitting a tool against a very narrow proportion of the Java platform. With that being said, we do not suggest *Gleipner* to be an end-all-be-all evaluation strategy for gadget chain detection tools. Rather, we envision a pipeline where tools attempt to tackle further challenges. Then, these improvements are first shown to be met on *Gleipner* and finally tested on real-world Java applications and dependencies.

(3) **Improper comparison of benchmarking results:** while developing *Gleipner*, we considered intricacies of all seven available tools to ensure that during the evaluation, none of them would suffer from **D3 unfair benchmarking of competitors**. For example, some tools use the heuristic of searching for gadget chains starting at trampoline methods rather than from *readObject()*. We ensured that both entry points are available to be discovered by the tools, apart from one category specifically designed to show this heuristic is being used (see Section 3). Also, we noticed that the tool *Crystallizer* only uses available constructors for payload construction (instead of manually

[14] see https://github.com/frohoff/ysoserial/blob/master/src/main/java/ysoserial/payloads/C3P0.java
[15] That would be ignoring the explanations for mismatches and counting the ▭ and ▭ in Table 5

iterating through properties). If we hadn't provided a constructor for all gadgets, the tool would have exhibited near to no findings at all. Conversely, this tool's deficiency is still visible in its inability to detect some of the replicated *Ysoserial* gadget chains.

(4) **Benchmarking omissions:** as mentioned in the introduction, **E1 not evaluating all contributions** is an issue that is prominently observed in *JDD*. While it is explained in a configuration file[16] that the claimed capability of resolving dynamic proxies was lost during refactoring, the tool should still have performed better on all other reflection test categories in *Gleipner*. This observation holds true even though we only had access to the tool's static analysis module. At most, the fuzzer would cull false positive findings. Going from *Ysoserial* would not have revealed the mismatch to the stated contribution of resolving reflection. Also, *Gleipner*, by design, contains only the labeled TP and FP gadget chains, thereby alleviating the benchmarking flaw **E3 false positives/negatives not tested**. Again, with *Ysoserial*, it is especially difficult to account for false negatives (and validating false positives is tedious).

We further go over the characteristics of good benchmarks in [54]. **Fairness** was already addressed above in (3), and in regard to **reproducibility**, providing our benchmark as a collection of JAR dependency files ensures the results are consistent over different environments. After all, all gadget chain detection tools operate on JAR files, so as long as the tool is set up to function properly in a different environment, it will consistently find the same gadget chains in the same JAR. Obviously, this is also beneficial for *Gleipner*'s **usability**. In theory, the only modification required for a tool to run our benchmark is to add the custom sink method signature. The usability is further enhanced by removing the time-consuming step of verifying detected false positives. Regarding **verifiability**, we wrote unit tests for all categories of gadget chains with the exception of those automatically generated (i.e., columns 3-5 in Table 4). These unit tests create a payload object for the gadget chain and serialize it into a byte array, from which it is then deserialized. This process executes a gadget chain as though it were deserialized from an insecure deserialization entry point. Finally, [54] states **relevance** as the most important trait of a benchmark. To this end, we cannot guarantee that *Gleipner* covers all possible scenarios and conditions required to be met for a gadget chain detection tool (which satisfies the benchmark) to be considered complete. However, it is a best-effort solution that accounts for all known limitations of such tools and Java program analysis in itself. Given how state-of-the-art algorithms performed on *Gleipner*, we believe it is still invaluable to bring research in Java gadget chain detection forward.

6 Related Work

6.1 Insecure Deserialization

Frohoff and Lawrence talk in 2015 [25] is seen as the starting point for research in *insecure deserialization*. From there, general research was conducted to further understand its intricacies. *Bechler* showed that *deserialization vulnerabilities* can not only be found in native Java serialization, but generally, every serialization library exposes similar types of vulnerabilities [2]. *Dietrich et al.* analyzed how self-looping gadget chains can be leveraged as a DoS attack vector [18]. Another area of research focuses on how to statically analyze Java through deserialization-aware call graph construction [48, 49]. *Sayar et al.* performed an in-depth study of how gadgets manifest themselves inside Java source code and whether or how they are later remediated [50]. Overall, this shows the complexity of *deserialization vulnerabilities*. Furthermore, the general research led to the development of various gadget chain detection tools [3, 9, 11, 12, 14, 15, 27, 31, 36, 44, 52, 58], half of which were published recently in 2023 and 2024. Again, this surge in research interest emphasizes the necessity of providing a comprehensive gadget chain benchmark.

[16] https://github.com/fdu-sec/JDD/blob/9dd3c3fccf71166791abbe0cc9c4cec6012f63c4/config/config.properties, line 46

6.2 Benchmarks

In regard to Java benchmarks, mentioning the DaCapo suite is mandatory [6]. While it probably is the most comprehensive benchmark at the time of writing, its main purpose is to provide realistic applications that researchers may use for performance measurement. In contrast, *Gleipner* is a benchmark for a specific use case in software and vulnerability analysis. As such, it specializes directly in evaluating gadget chain detection algorithms. This is a task not covered by static analysis benchmarks [1, 7, 53], automated program repair (e.g., the BEARS suite [37]), or vulnerability benchmarks [5, 7, 41].

7 Limitations

Gleipner is a benchmark for gadget chain detection algorithms in Java. On one hand, this means it is not directly applicable to the evaluation of similar tools in other programming languages, such as [51] (*C#*) or [17, 43] (*PHP*). On the other hand, it does not serve as a benchmark for insecure deserialization entry points. A benchmark for this use case has already been developed in [47]. Furthermore, *Gleipner* is based on known limitations to program analysis and gadget chain detection. Thus, we neither claim that our benchmark is entirely complete nor that a gadget chain detection tool fully satisfying our benchmark is guaranteed to be sound and precise. However, going from the subpar performance of state-of-the-art tools on *Gleipner* (Section 4.3) and the lack of a benchmark to begin with, we believe this limitation to be negligible. Also, *Gleipner's* modular design makes it easy to extend with further test cases if necessary.

8 Conclusion

In this work, we designed and built *Gleipner*, a benchmark for gadget chain detection algorithms. It probes eight challenges in finding gadget chains (see Section 2), including emulations of large search space, complex payload construction, and a thorough representation of the *Java Reflection API*. Using *Gleipner*, we evaluated all openly available gadget chain detection algorithms. The results show that none of these closely satisfy all challenges and also that previously relying on *Ysoserial* as a benchmark has led to severe benchmarking flaws. With our benchmark, we now provide a transparent and streamlined evaluation methodology, and, moreover, it sets a target for novel gadget chain detection tools.

9 Data Availability

We anonymously share the source code of *Gleipner* and tool instrumentation (Section 4) including a usage explanation with the link **https://github.com/software-engineering-and-security/Gleipner**.

Acknowledgments

This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

References

[1] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. *ACM SIGPLAN Notices* 49, 6 (June 2014), 259–269. https://doi.org/10.1145/2666356.2594299

[2] Moritz Bechler. 2017. Java Unmarshaller Security. (May 2017). https://raw.githubusercontent.com/mbechler/marshalsec/master/marshalsec.pdf

[3] Moritz Bechler. 2017. mbechler/serianalyzer. https://github.com/mbechler/serianalyzer original-date: 2016-02-28T09:34:31Z.

[4] Guru Bhandari, Amara Naseer, and Leon Moonen. 2021. CVEfixes: automated collection of vulnerabilities and their fixes from open-source software. In *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE 2021)*. Association for Computing Machinery, New York, NY, USA, 30–39. https://doi.org/10.1145/3475960.3475985

[5] Paul E. Black. 2018. A Software Assurance Reference Dataset: Thousands of Programs With Known Bugs. *Journal of Research of the National Institute of Standards and Technology* 123 (April 2018), 1–3. https://doi.org/10.6028/jres.123.005

[6] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications* (Portland, OR, USA). ACM Press, New York, NY, USA, 169–190. https://doi.org/10.1145/1167473.1167488

[7] Tim Boland and Paul E. Black. 2012. Juliet 1.1 C/C++ and Java Test Suite. *Computer* 45, 10 (Oct. 2012), 88–90. https://doi.org/10.1109/MC.2012.345 Conference Name: Computer.

[8] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. 2002. ASM: A code manipulation tool to implement adaptable systems. (Jan. 2002).

[9] Luca Buccioli, Stefano Cristalli, Edoardo Vignati, Lorenzo Nava, Daniele Badagliacca, Danilo Bruschi, Long Lu, and Andrea Lanzi. 2023. JChainz: Automatic Detection of Deserialization Vulnerabilities for the Java Language. In *Security and Trust Management: 18th International Workshop, STM 2022, Copenhagen, Denmark, September 29, 2022, Proceedings*. Springer-Verlag, Berlin, Heidelberg, 136–155. https://doi.org/10.1007/978-3-031-29504-1_8

[10] Quang-Cuong Bui, Riccardo Scandariato, and Nicolás E. Díaz Ferreyra. 2022. Vul4J: a dataset of reproducible Java vulnerabilities geared towards the study of program repair techniques. In *Proceedings of the 19th International Conference on Mining Software Repositories (MSR '22)*. Association for Computing Machinery, New York, NY, USA, 464–468. https://doi.org/10.1145/3524842.3528482

[11] Sicong Cao, Biao He, Xiaobing Sun, Yu Ouyang, Chao Zhang, Xiaoxue Wu, Ting Su, Lili Bo, Bin Li, Chuanlei Ma, Jiajia Li, and Tao Wei. 2023. ODDFuzz: Discovering Java Deserialization Vulnerabilities via Structure-Aware Directed Greybox Fuzzing. IEEE Computer Society, 2726–2743. https://doi.org/10.1109/SP46215.2023.10179377

[12] Sicong Cao, Xiaobing Sun, Xiaoxue Wu, Lili Bo, Bin Li, Rongxin Wu, Wei Liu, Biao He, Yu Ouyang, and Jiajia Li. 2023. Improving Java Deserialization Gadget Chain Mining via Overriding-Guided Object Generation. In *Proceedings of the 45th International Conference on Software Engineering (ICSE '23)*. IEEE Press, 397–409. https://doi.org/10.1109/ICSE48619.2023.00044

[13] Byeong-Mo Chang and Kwanghoon Choi. 2016. A review on exception analysis. *Information and Software Technology* 77 (Sept. 2016), 1–16. https://doi.org/10.1016/j.infsof.2016.05.003

[14] Bofei Chen, Lei Zhang, Xinyou Huang, Yinzhi Cao, Keke Lian, Yuan Zhang, and Min Yang. 2024. Efficient Detection of Java Deserialization Gadget Chains via Bottom-up Gadget Search and Dataflow-aided Payload Construction. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 150–150.

[15] Xingchen Chen, Baizhu Wang, Ze Jin, Yun Feng, Xianglong Li, Xincheng Feng, and Qixu Liu. 2023. Tabby: Automated Gadget Chain Detection for Java Deserialization Vulnerabilities. In *2023 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 179–192. https://doi.org/10.1109/DSN58367.2023.00028 ISSN: 2158-3927.

[16] Hung-Wei Chiang and Chao-Lung Chou. 2024. JAVA Web System Deserialization Vulnerability Detection Technology. *Communications of the CCISA* 30, 2 (May 2024), 45–63. https://cccisa.ccisa.org.tw/article/view/3049 Number: 2.

[17] Johannes Dahse, Nikolai Krein, and Thorsten Holz. 2014. Code Reuse Attacks in PHP: Automated POP Chain Generation. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS '14)*. Association for Computing Machinery, New York, NY, USA, 42–53. https://doi.org/10.1145/2660267.2660363

[18] Jens Dietrich, Kamil Jezek, Shawn Rasheed, Amjed Tahir, and Alex Potanin. 2017. Evil Pickles: DoS Attacks Based on Object-Graph Engineering. https://doi.org/10.4230/LIPIcs.ECOOP.2017.10

[19] Ben Evans. 2020. Behind the scenes: How do lambda expressions really work in Java? https://blogs.oracle.com/javamagazine/post/behind-the-scenes-how-do-lambda-expressions-really-work-in-java

[20] federicodotta, Jeremy Goldstein, and András Veres-Szentikrályi. 2021. Java Deserialization Scanner. https://github.com/federicodotta/Java-Deserialization-Scanner original-date: 2015-12-08T14:31:15Z.

[21] Apache Foundation. 2024. apache/groovy: Apache Groovy: A powerful multi-faceted programming language for the JVM platform. https://github.com/apache/groovy/tree/master

[22] OWASP Foundation. 2021. OWASP Top Ten. Retrieved 2023-10-23 from https://owasp.org/www-project-top-ten/

[23] George Fourtounis, George Kastrinis, and Yannis Smaragdakis. 2018. Static analysis of Java dynamic proxies. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2018)*. Association for Computing Machinery, New York, NY, USA, 209–220. https://doi.org/10.1145/3213846.3213864

[24] Chris Frohoff. 2023. *ysoserial*. https://github.com/frohoff/ysoserial original-date: 2015-01-28T07:13:55Z.

[25] Chris Frohoff and Gabriel Lawrence. 2015. AppSecCali 2015: Marshalling Pickles by frohoff. https://frohoff.github.io/appseccali-marshalling-pickles/
[26] Pierre Graux, Jean-François Lalande, Valérie Viet Triem Tong, and Pierre Wilke. 2021. Preventing serialization vulnerabilities through transient field detection. In *Proceedings of the 36th Annual ACM Symposium on Applied Computing (SAC '21)*. Association for Computing Machinery, New York, NY, USA, 1598–1606. https://doi.org/10.1145/3412841.3442033
[27] Ian Haken. 2018. Automated Discovery of Deserialization Gadget Chains. https://i.blackhat.com/us-18/Thu-August-9/us-18-Haken-Automated-Discovery-of-Deserialization-Gadget-Chains.pdf
[28] Hayyan Salman Hasan, Hasan Muhammad Deeb, and Behrouz Tork Ladani. 2023. A Machine Learning Approach for Detecting and Categorizing Sensitive Methods in Android Malware. *The ISC International Journal of Information Security* 15, 1 (Jan. 2023), 59–71. https://doi.org/10.22042/isecure.2022.321436.741 Publisher: Iranian Society of Cryptology.
[29] Nikolaos Koutroumpouchos, Georgios Lavdanis, Eleni Veroni, Christoforos Ntantogian, and Christos Xenakis. 2019. ObjectMap: detecting insecure object deserialization. In *Proceedings of the 23rd Pan-Hellenic Conference on Informatics (PCI '19)*. Association for Computing Machinery, New York, NY, USA, 67–72. https://doi.org/10.1145/3368640.3368680
[30] Bruno Kreyssig and Alexandre Bartel. 2024. Analyzing Prerequistes of known Deserializtion Vulnerabilities on Java Applications. In *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering (EASE '24)*. Association for Computing Machinery, New York, NY, USA, 28–37. https://doi.org/10.1145/3661167.3661176
[31] Zhaojia Lai, Haipeng Qu, and Lingyun Ying. 2022. A Composite Discover Method for Gadget Chains in Java Deserialization Vulnerability. virtual.
[32] Davy Landman, Alexander Serebrenik, and Jurgen J. Vinju. 2017. Challenges for static analysis of Java reflection: literature review and empirical study. In *Proceedings of the 39th International Conference on Software Engineering (ICSE '17)*. IEEE Press, Buenos Aires, Argentina, 507–518. https://doi.org/10.1109/ICSE.2017.53
[33] Li Li, Tegawendé F. Bissyandé, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein, and Le Traon. 2017. Static analysis of android apps: A systematic literature review. *Information and Software Technology* 88 (Aug. 2017), 67–95. https://doi.org/10.1016/j.infsof.2017.04.001
[34] Siliang Li and Gang Tan. 2014. Exception analysis in the Java Native Interface. *Science of Computer Programming* 89 (Sept. 2014), 273–297. https://doi.org/10.1016/j.scico.2014.01.018
[35] Weicheng Li, Hui Lu, Yanbin Sun, Shen Su, Jing Qiu, and Zhihong Tian. 2023. Improving Precision of Detecting Deserialization Vulnerabilities with Bytecode Analysis. In *2023 IEEE/ACM 31st International Symposium on Quality of Service (IWQoS)*. 1–2. https://doi.org/10.1109/IWQoS57198.2023.10188756 ISSN: 2766-8568.
[36] Yifan Luo and Baojiang Cui. 2024. Rev Gadget: A Java Deserialization Gadget Chains Discover Tool Based on Reverse Semantics and Taint Analysis. In *Advances in Internet, Data & Web Technologies*, Leonard Barolli (Ed.). Springer Nature Switzerland, Cham, 229–240. https://doi.org/10.1007/978-3-031-53555-0_22
[37] Fernanda Madeiral, Simon Urli, Marcelo Maia, and Martin Monperrus. 2019. BEARS: An Extensible Java Bug Benchmark for Automatic Program Repair Studies. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 468–478. https://doi.org/10.1109/SANER.2019.8667991 ISSN: 1534-5351.
[38] MITRE. 2006. CWE - CWE-502: Deserialization of Untrusted Data (4.14). https://cwe.mitre.org/data/definitions/502.html
[39] MvnRepository. [n. d.]. Maven Repository: Search/Browse/Explore. https://mvnrepository.com/
[40] Oracle. 2024. Java™ Platform, Standard Edition 11 API Specification. https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/ReflectiveOperationException.html
[41] OWASP. [n. d.]. OWASP Benchmark | OWASP Foundation. https://owasp.org/www-project-benchmark/
[42] Rohan Padhye, Caroline Lemieux, and Koushik Sen. 2019. JQF: coverage-guided property-based testing in Java. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 398–401. https://doi.org/10.1145/3293882.3339002
[43] Sunnyeo Park, Daejun Kim, Suman Jana, and Sooel Son. 2022. FUGIO: Automatic Exploit Generation for PHP Object Injection Vulnerabilities. In *31st USENIX Security Symposium (USENIX Security 22)*. 197–214. https://www.usenix.org/conference/usenixsecurity22/presentation/park-sunnyeo
[44] Shawn Rasheed and Jens Dietrich. 2021. A hybrid analysis to detect Java serialisation vulnerabilities. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20)*. Association for Computing Machinery, New York, NY, USA, 1209–1213. https://doi.org/10.1145/3324884.3418931
[45] Siegfried Rasthofer, Steven Arzt, and Eric Bodden. 2014. A Machine-learning Approach for Classifying and Categorizing Android Sources and Sinks. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*. The Internet Society. https://www.ndss-symposium.org/ndss2014/machine-learning-approach-classifying-and-categorizing-android-sources-and-sinks
[46] Roger Riggs. 2016. JEP 290: Filter Incoming Serialization Data. https://openjdk.org/jeps/290

[47] Alessandro Sabatini. 2020. Evaluating the Testability of Insecure Deserialization Vulnerabilities via Static Analysis. (2020). https://hdl.handle.net/10589/187947

[48] Joanna C. S. Santos, Reese A. Jones, Chinomso Ashiogwu, and Mehdi Mirakhorli. 2021. Serialization-aware call graph construction. In *Proceedings of the 10th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis (SOAP 2021)*. Association for Computing Machinery, New York, NY, USA, 37–42. https://doi.org/10.1145/3460946.3464319

[49] Joanna C. S. Santos, Mehdi Mirakhorli, and Ali Shokri. 2024. Seneca: Taint-Based Call Graph Construction for Java Object Deserialization. *Proceedings of the ACM on Programming Languages* 8, OOPSLA1 (April 2024), 134:1125–134:1153. https://doi.org/10.1145/3649851

[50] Imen Sayar, Alexandre Bartel, Eric Bodden, and Yves Le Traon. 2023. An In-depth Study of Java Deserialization Remote-Code Execution Exploits and Vulnerabilities. *ACM Transactions on Software Engineering and Methodology* 32, 1 (Feb. 2023), 25:1–25:45. https://doi.org/10.1145/3554732

[51] Mikhail Shcherbakov and Musard Balliu. 2021. SerialDetector: Principled and Practical Exploration of Object Injection Vulnerabilities for the Web. In *Network and Distributed Systems Security (NDSS) Symposium 202121-24 February 2021*. https://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-288522

[52] Prashast Srivastava, Flavio Toffalini, Kostyantyn Vorobyov, François Gauthier, Antonio Bianchi, and Mathias Payer. 2023. Crystallizer: A Hybrid Path Analysis Framework to Aid in Uncovering Deserialization Vulnerabilities. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 1586–1597. https://doi.org/10.1145/3611643.3616313

[53] Li Sui, Jens Dietrich, Michael Emery, Shawn Rasheed, and Amjed Tahir. 2018. On the Soundness of Call Graph Construction in the Presence of Dynamic Language Features - A Benchmark and Tool Evaluation. In *Programming Languages and Systems (Lecture Notes in Computer Science)*, Sukyoung Ryu (Ed.). Springer International Publishing, Cham, 69–88. https://doi.org/10.1007/978-3-030-02768-1_4

[54] Jóakim v. Kistowski, Jeremy A. Arnold, Karl Huppler, Klaus-Dieter Lange, John L. Henning, and Paul Cao. 2015. How to Build a Benchmark. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering (ICPE '15)*. Association for Computing Machinery, New York, NY, USA, 333–336. https://doi.org/10.1145/2668930.2688819

[55] Erik van der Kouwe, Gernot Heiser, Dennis Andriesse, Herbert Bos, and Cristiano Giuffrida. 2019. SoK: Benchmarking Flaws in Systems Security. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. 310–325. https://doi.org/10.1109/EuroSP.2019.00031

[56] Kostyantyn Vorobyov, François Gauthier, Sora Bae, Padmanabhan Krishnan, and Rebecca O'Donoghue. 2022. Synthesis of Java Deserialisation Filters from Examples. In *2022 IEEE 46th Annual Computers, Software, and Applications Conference (COMPSAC)*. 736–745. https://doi.org/10.1109/COMPSAC54236.2022.00123 ISSN: 0730-3157.

[57] Fengguo Wei, Xingwei Lin, Xinming Ou, Ting Chen, and Xiaosong Zhang. 2018. JN-SAF: Precise and Efficient NDK/JNI-aware Inter-language Static Analysis Framework for Security Vetting of Android Applications with Native Code. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18)*. Association for Computing Machinery, New York, NY, USA, 1137–1150. https://doi.org/10.1145/3243734.3243835

[58] Junjie Wu, Jingling Zhao, and Junsong Fu. 2022. A Static Method to Discover Deserialization Gadget Chains in Java Programs. In *Proceedings of the 2022 2nd International Conference on Control and Intelligent Robotics (ICCIR '22)*. Association for Computing Machinery, New York, NY, USA, 800–805. https://doi.org/10.1145/3548608.3559310

[59] Quan Zhang, Yiwen Xu, Zijing Yin, Chijin Zhou, and Yu Jiang. 2024. Automatic Policy Synthesis and Enforcement for Protecting Untrusted Deserialization. In *Proceedings 2024 Network and Distributed System Security Symposium*. Internet Society, San Diego, CA, USA. https://doi.org/10.14722/ndss.2024.24053

Received 2024-08-22; accepted 2025-01-14