# Automatic Policy Synthesis and Enforcement for Protecting Untrusted Deserialization

Quan Zhang, Yiwen Xu, Zijing Yin, Chijin Zhou, and Yu Jiang*
BNRist, School of Software, Tsinghua University

***Abstract*—Java deserialization vulnerabilities have long been a grave security concern for Java applications. By injecting malicious objects with carefully crafted structures, attackers can reuse a series of existing methods during deserialization to achieve diverse attacks like remote code execution. To mitigate such attacks, developers are encouraged to implement policies restricting the object types that applications can deserialize. However, the design of precise policies requires expertise and significant manual effort, often leading to either the absence of policy or the implementation of inadequate ones.**

**In this paper, we propose DESERIGUARD, a tool designed to assist developers in securing their applications seamlessly against deserialization attacks. It can automatically formulate a policy based on the application's semantics and then enforce it to restrict illegal deserialization attempts. First, DESERIGUARD utilizes dataflow analysis to construct a semantic-aware property tree, which records the potential structures of deserialized objects. Based on the tree, DESERIGUARD identifies the types of objects that can be safely deserialized and synthesizes an allowlist policy. Then, with the Java agent, DESERIGUARD can seamlessly enforce the policy during runtime to protect various deserialization procedures. In evaluation, DESERIGUARD successfully blocks all deserialization attacks on 12 real-world vulnerabilities. In addition, we compare DESERIGUARD's automatically synthesized policies with 109 developer-designed policies. The results demonstrate that DESERIGUARD effectively restricts 99.12% more classes. Meanwhile, we test the policy-enhanced applications with their unit tests and integration tests, which demonstrate that DESERIGUARD's policies will not interfere with applications' execution and induce a negligible time overhead of 2.17%.**

## I. INTRODUCTION

Serialization and deserialization are essential mechanisms offered by programming languages such as Java, JavaScript, PHP, and C# for object and byte stream transformation. These mechanisms facilitate the seamless transmission of objects across networks and their storage in databases [1]. However, these capabilities also introduce a type of significant vulnerability, which can cause severe consequences such as remote code execution (RCE), denial of service (DoS), and server-side request forgery (SSRF). Over the past five years, approximately 800 vulnerabilities collected by Common Vulnerabilities and Exposures (CVE) [2] are related to the Common Weakness Enumeration (CWE) 502 [3], which represents the deserialization of untrusted data.

The root cause of deserialization vulnerability is that an application tries to deserialize an untrusted object without protection, allowing attackers to control the execution flow of the applications during the deserialization process. Specifically, attackers begin by carefully constructing a nested object with classes in the application's classpath. This nested object possesses a precisely designed structure, in which its properties are assigned specific objects. During the deserialization process of the nested object, the application is forced to recursively deserialize its properties in the desired order of attackers, which leads to the execution of a series of methods that form an exploitation gadget chain. Such a gadget chain could finally lead to the execution of some security-sensitive methods such as `Runtime.exec()`, causing malicious impacts like RCE [4]. However, if we prevent the deserialization of any classes involved in the gadget chain, we can effectively defend applications against deserialization attacks.

Therefore, at present, a prevailing approach for mitigating deserialization attacks is to examine the currently deserialized object and verify its legitimacy. Developers have long been advised to set appropriate deserialization policies for blocking potentially malicious types to secure their applications. For example, in 2016, Java Enhancement Proposal (JEP) 290 [5] introduced a new feature that allows users to specify a policy for a deserialization entry. Popular deserialization libraries, such as XStream and FastJson [6], [7], also provide flexible interfaces for setting blocklist or allowlist policies. Nevertheless, up to this point, the community is still plagued by deserialization attacks.

The primary obstacle preventing developers from adequately protecting their applications stems from the challenge of formulating fine-grained deserialization policies. Firstly, since setting policies requires significant human effort for debugging and maintenance, many developers, who lack sufficient security awareness, choose not to set policies. Secondly, some applications employ a blocklist policy by collecting known exploitation gadgets. Nevertheless, attackers continually discover new exploitable gadgets to pursue malicious intentions, rendering such blocklists inadequate over time. For example, in CVE-2017-1000353 [8], attackers successfully bypassed Jenkins's blocklist and achieved an RCE attack. Therefore, the allowlist policy containing limited permitted classes is recognized as the most effective approach to safeguarding applications' deserialization process. However, designing an accurate allowlist policy demands laborious manual efforts and is prone to errors. Thus, many applications opt for a loose policy to avoid interrupting applications' normal execution,

* Yu Jiang is the corresponding author.

Network and Distributed System Security (NDSS) Symposium 2024
26 February - 1 March 2024, San Diego, CA, USA
ISBN 1-891562-93-2
https://dx.doi.org/10.14722/ndss.2024.24053
www.ndss-symposium.org

which gives attackers opportunities to bypass existing defenses. For instance, the application Ofbiz sets a policy and allows deserialization of all classes whose names match "java\..*", which can be bypassed by attackers and causes the vulnerability CVE-2021-26295 [9].

To save developers' efforts, this paper proposes DESERIGUARD, a framework designed to automate the synthesis and enforcement of deserialization policies. The key insight behind DESERIGUARD is that the types of deserialized objects can be deduced from program semantics. Following the insight, DESERIGUARD first traces the dataflow of the deserialized object and its properties (i.e., member variables) to deduce their object types based on applications' semantics, finally constructing the Semantic-Aware Property Tree (SAPT). This tree records the possible structures of potentially deserialized objects. Based on the tree, DESERIGUARD identifies all possible classes that should be permitted for deserialization and synthesizes an allowlist policy. Next, DESERIGUARD automatically resolves the auditing positions where deserialization libraries decode the type information from the byte stream of serialized objects. After identifying the positions, DESERIGUARD leverages a Java agent to enforce the policy on the objects that are being deserialized. Finally, with DESERIGUARD, developers can secure their applications' deserializations with automatically synthesized and enforced allowlist policies.

We comprehensively evaluated DESERIGUARD using 12 real-world vulnerabilities and 109 developer-designed policies. Specifically, DESERIGUARD successfully resists the deserialization attack on all 12 vulnerabilities with only 2.17% of runtime overhead on average. Meanwhile, we carry out extensive testing to verify that DESERIGUARD will not interfere with the normal execution of policy-enhanced applications, which can pass all unit tests and integration tests in their projects. Furthermore, we conduct a comparison between DESERIGUARD's policies and 109 policies that are formulated by developers of popular applications. Our findings reveal that, on average, DESERIGUARD's policies restrict the deserialization of 99.12% more classes than the developer-defined policies. We also compare DESERIGUARD with a state-of-the-art deserialization defense tool proposed by Cristalli et al. [10], where DESERIGUARD provides identical defense performance with no false alarms.

Our contributions are listed as follows.

- We propose an automatic policy generation and enforcement framework to safeguard the deserialization processes of Java applications.
- We implement DESERIGUARD, which synthesizes the allowlist policy based on the Semantic-Aware Property Tree and seamlessly enforces the policy for different deserialization libraries.
- We evaluate the DESERIGUARD on 12 real-world vulnerabilities and compare the generated policies with 109 developer-designed policies. The results show that DESERIGUARD can synthesize more fine-grained policies and will not interfere with applications' normal execution.

## II. BACKGROUND AND MOTIVATION

### A. Deserialization Vulnerability

Serialization and deserialization are fundamental mechanisms offered by numerous popular programming languages, such as Java, JavaScript, and PHP. These mechanisms enable the transformation between objects and byte streams, making them widely employed for network transmission and persistent storage of objects. Meanwhile, with different deserialization libraries, developers can transform objects into byte streams in various formats, such as JSON through the *FastJson* library and XML through the *XStream* library. However, according to the Common Weakness Enumeration (CWE) 502 [3], attackers can perform RCE or DoS attacks from the deserialization entries if developers do not restrict the object classes to be deserialized. The deserialization vulnerability has constantly been a severe threat. Over the past five years, approximately 800 vulnerabilities related to deserialization have been registered in CVEs [2].

The exploitation of a Java deserialization vulnerability begins with an exposed deserialization entry, like the `readObject` method included in JDK [11] and the `parse` method in the *FastJson* library [6]. These methods take a byte stream as input and return the deserialized object. Given that deserialization is commonly used in network transmission, many deserialization entries have to parse byte streams that may come from untrusted sources, providing extensive attack surfaces for attackers.

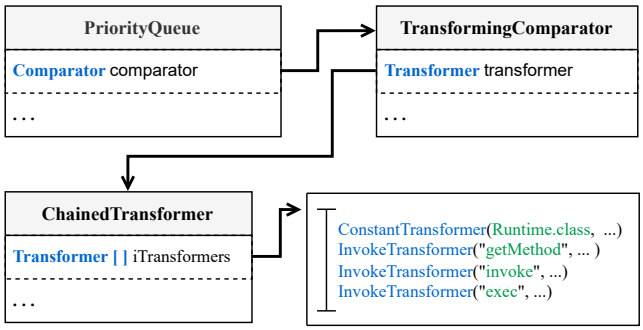


Fig. 1. A malicious nested object crafted by attackers. This object's deserialization can trigger a gadget chain that performs an RCE attack.

After finding an exposed deserialization entry, attackers will send a malicious byte stream to the target application for deserialization. Such a malicious byte stream is serialized from a carefully structured nested object. To reconstruct such an object from the malicious byte stream, the victim application needs to recursively deserialize the objects of its properties, resulting in the invocation of a sequence of methods. Therefore, by controlling the structure of the nested object, attackers can connect a series of methods to a chain, named **gadget chain**, and hijack the victim application to execute such a chain. Notably, all these methods on gadget chains must belong to the classes in the victim application's classpath, allowing the application to find their definitions and invoke them.

Figure 1 depicts an example of a nested object. The object is of the class `PriorityQueue`, where its `comparator` property is assigned a `TransformingComparator` object. Within the object, there exists a property of type

`Transformer`, which attackers can exploit by setting it as an object of the `ChainedTransformer` class. Next, the `iTransformers` property of the `ChainedTransformer` class is an array of `Transformer` objects that comprises multiple transformers. By skillfully assembling the array with `ConstantTranformer` and `InvokeTransformer`, attackers can create a `Runtime` object and invoke the `exec` method during the deserialization, thereby achieving remote code execution (RCE) attacks.

The procedure of deserializing the malicious object in Figure 1 is introduced as follows. First, in order to correctly deserialize `PriorityQueue` and its elements, the victim application needs to compare these elements to establish their order using the `comparator` property of `PriorityQueue`. Next, to deserialize the `TransformingComparator` object assigned to `comparator`, the application has to deserialize its `transformer` property recursively. As the `transformer` property is a `ChainedTransformer` that contains a transformer array `iTransformers`, the victim application has to deserialize the array's all elements, which utilizes the `ConstantTranformer` to create a `Runtime` object and `InvokeTransformer` to invoke its `exec` method for arbitrary commands execution. At this point, the deserialization process is not completed, and the reconstruction of the nested object is not finished. However, the gadget chain has already been triggered, allowing attackers to achieve their objectives.

### B. Deserialization Policy

The community has made significant efforts toward deserialization protection. The most common defense method is enforcing a policy to restrict the deserialization of specific object types. In this way, certain classes used in gadget chains are forbidden to be deserialized, effectively restricting exploitation attempts. To support such defense mechanisms, many popular deserialization libraries offer customization options for blocklist-based or allowlist-based deserialization policies. For instance, *XStream* [7] has supported policy enforcement since version 1.4.7, released in 2014. In 2016, Java Enhancement Proposal (JEP) 290 [5] provided an `ObjectInputFilter` class to set policy for the `ObjectInputStream` class in JDK. In addition to supporting customizing, *FastJson* [6] and *Jackson* [12] feature an in-built blocklist mechanism to restrict the deserialization of classes on known gadget chains.

```
"byte\\[\\]", "foo", "\\[Z", "\\[B",
"\\[S", "\\[I", "\\[J", "\\[F", "\\[D", "\\[C",
"SerializationInjector",
"java\..*",
"sun\.util\.calendar\..*",
"org\.apache\.ofbiz\..*",
"org\.codehaus\.groovy\.runtime\.GStringImpl",
"groovy\.lang\.GString",
```

Fig. 2. Developer-designed allowlist policy of the Ofbiz project. The policy is defined in the patch for CVE-2019-0189.

With the support of new security features in libraries, developers are encouraged to customize the deserialization policies of their applications according to their requirements. The most common ones are blocklists containing classes in all known exploitation gadgets collected by the community [6], [13], [14]. For example, in the Jenkins project [15], a famous automation server with over 20k stars on GitHub, developers formulate a blocklist including exploitable classes in its classpath, such as `CommonsCollection` and `CommonsBeanutils`. With such a policy, Jenkins can deny the deserialization of `TransformingComparator` and `InvokeTransformer`, effectively blocking the gadget chain in Figure 1. Nevertheless, attackers are continuously mining new exploitation gadget chains to bypass blocklists [16], [17]. In CVE-2017-1000353 [8], `SignedObject` is found exploitable but is not included in the blocklist.

Hence, allowlist policies that include necessary classes are regarded as a more effective approach for applications to defend against deserialization attacks. However, as analyzed by Imen et al. [18], developers often assume that objects to be deserialized are of type `Object` or `Serializable`, which are the superclasses of all serializable classes. This implies that developers are unaware of the concrete classes of deserialized objects and are unable to formulate strict allowlists when developing the application. Therefore, they usually provide a loose allowlist policy to prevent mistakenly blocking necessary classes. As demonstrated by Ofbiz's policy in Figure 2, developers may allow the deserialization of all developer-defined classes in the application, which can be matched using the pattern like "org\.apache\.ofbiz\..*", and permit classes within JDK, denoted by regex pattern "java\..*".

### C. Motivation Example

Despite Java deserialization being a persistent and significant threat, numerous applications still lack a strict policy to mitigate this risk. On GitHub, over 3,600 Java projects with 100+ stars make use of deserialization mechanisms offered by libraries such as *FastJson*, *XStream*, and JDK's *ObjectInputStream*. However, out of these projects, only around 340 have implemented policies specifically tailored to their deserialization processes. The primary reason for this situation is that configuring policies often demands extensive manual efforts from developers, resulting in many developers, who lack adequate security awareness, choosing not to prioritize defense against deserialization attacks.

```
1  protected Class<?> resolveClass(ObjectStreamClass classDesc)
                    throws IOException, ClassNotFoundException {
2      String className = classDesc.getName();
3      // BlockList exploits; eg: don't allow RMI here
4      if (className.contains("java.rmi.server")) {
5          Debug.logWarning("***Incompatible class***");
6          return null;
7      }                                   Patch for CVE-2021-26295
8      if (!whitelistPattern.matcher(className).find()) {
9          Debug.logWarning("***Incompatible class***");
10         throw new ClassCastException("Incompatible class");
11     }                                   Patch for CVE-2019-0189
12     return ObjectType.loadClass(classDesc.getName());
13 }
```

Fig. 3. Patches on Ofbiz for CVE-2019-0189 and CVE-2021-26295.

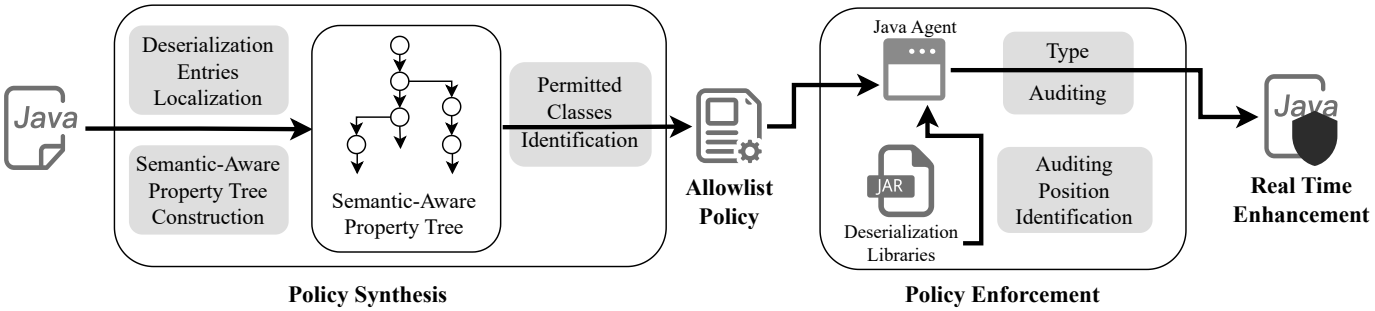


Fig. 4. Overview of DESERIGUARD. DESERIGUARD consists of two modules: a Policy Synthesis module for designing allowlist policy for each deserialization entry to encompass potential types that could be deserialized, and a Policy Enforcement module for real-time type auditing of untrusted deserialization.

Furthermore, though developers begin formulating policies in response to attacks on their applications, manually designed policies remain prone to errors due to inadequate blocklists or overly permissive allowlists. In detail, Ofbiz first exposed a vulnerable deserialization entry with no defense at all, leading to RCE attacks. Such vulnerability was assigned as CVE-2019-0189 [19] and patched with an allowlist policy, as depicted in lines 8∼11 of Figure 3. The `resolveClass` method of `ObjectInputStream` is commonly overridden by developers to validate the safety of the currently deserialized object's class. In this approach, Ofbiz employed an allowlist policy, shown in Figure 2, to effectively prevent the deserialization of gadget chains, including the chain depicted in Figure 1.

However, for convenience, the developers of Ofbiz allowed deserialization of all classes whose names match "java\..*", inadvertently enabling attackers to bypass the allowlist policy. With class `java.rmi.server.RemoteObject`, attackers can achieve RCE through RMI attacks [20], [21]. Therefore, CVE-2021-26295 [9] was subsequently exposed and patched with a blocklist in line 4∼7 of Figure 3. The blocklist additionally restricted the deserialization of classes whose names start with "java.rmi.server". Unfortunately, such a blocklist was soon proved to be insufficient in CVE-2021-29200 [22], as attackers can utilize `RMIConnectionImpl_Stub` class with a different prefix to bypass the blocklist. To fix the vulnerability, developers expanded the blocklist in line 4 of Figure 3 with a prefix "java.rmi".

Although the policy was sufficient after several patches, Ofbiz was still vulnerable due to the improper implementation of policy enforcement. When matching the allowlist in line 8 of Figure 3, Ofbiz utilized a regular match that only necessitated the presence of the pattern string within the target string. In CVE-2021-30128 [23], attackers can bypass the regular match by crafting a particular string of class names and thus deserialize arbitrary classes. After applying multiple patches and addressing four CVEs, the developers of Ofbiz finally developed an appropriate policy for this deserialization entry and successfully enforced it.

**Challenges.** During the procedure of fixing the above vulnerabilities, we find two main challenges that impede developers from establishing a strict policy. Firstly, formulating a deserialization policy requires extensive manual effort and expert experience, making it prone to errors. Second, different deserialization libraries need various adaptations, which could be incorrectly implemented when enforcing a policy. To address the first challenge, DESERIGUARD utilizes dataflow analysis to construct a Semantic-Aware Property Tree to automatically synthesize a policy, thereby designing a more stringent allowlist policy and reducing the need for manual efforts. Moreover, by automatically identifying auditing positions in different deserialization libraries, DESERIGUARD seamlessly enforces the policy before triggering the exploitation gadget, ensuring the strict auditing of the objects to be deserialized.

## III. THREAT MODEL

DESERIGUARD is designed to provide seamless protection against deserialization attacks for developers. Its threat model operates under the assumption that attackers can access a deserialization entry within the target application and manipulate the input provided to that entry. Meanwhile, we consider that attackers have knowledge of all classes and their member methods within the victim applications' classpaths, enabling them to construct malicious gadget chains to accomplish their attack objectives. To mitigate such attacks, developers should supply the application's source code for DESERIGUARD's analysis and need to launch the Java agent when the application starts. In addition, DESERIGUARD requires an uncompromised machine and a trusted Java Virtual Machine (JVM) execution environment as its foundation. Consequently, attacks originating from other aspects, such as vulnerabilities in the operating system kernel or memory errors in the JVM, fall outside the scope of DESERIGUARD's protection. Moreover, if attackers can find exploitable gadgets within the classes that developers need to deserialize, DESERIGUARD cannot prevent the attack.

## IV. METHODOLOGY

As shown in Figure 4, to secure a Java application with an allowlist-based policy, DESERIGUARD needs to first synthesize a policy with *Policy Synthesis* module and then enforces the policy with *Policy Enforcement* module. In Policy Synthesis, DESERIGUARD constructs a Semantic-Aware Property Tree (SAPT) with dataflow analysis. Based on the tree, DESERIGUARD deduces all the possible types of deserialized objects and formulates an allowlist policy. Following this, DESERIGUARD performs byte code instrumentation with a Java agent to enforce the policy on different deserialization libraries. Finally, developers can efficiently enhance their applications without laborious analysis and tedious adaptation.

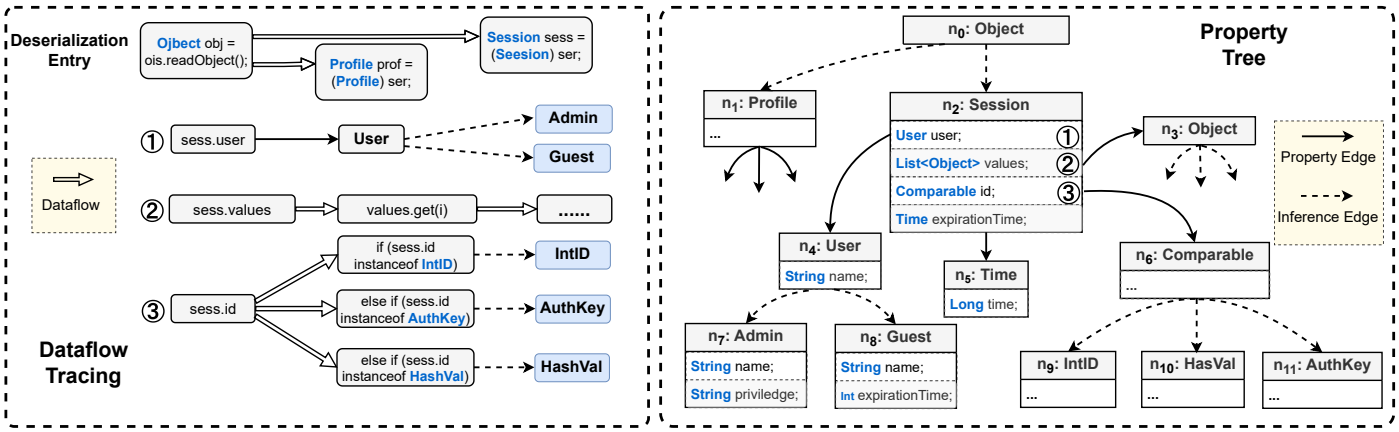


Fig. 5. An example of Semantic-Aware Property Tree (SAPT) construction. By tracing the dataflow along the properties of a deserialized object and identifying their possible types, a SAPT is established to describe the potential structure of deserialized objects. The dataflow of the program is indicated by double-lined arrows. SAPT has two types of edges: property edges with solid-lined arrows and inference edges with dotted arrows. The source code of the example is shown in Listing 1 in the Appendix.

### A. Policy Synthesis

In Policy Synthesis, DESERIGUARD is responsible for synthesizing an allowlist policy for each deserialization entry to encompass the potential types that could be deserialized. In detail, for an application, DESERIGUARD can resolve the types that a deserialized object and its properties are desired to be according to the program's semantics. By incorporating these classes into the allowlist policy, DESERIGUARD can mitigate the deserialized attack while still enabling the application to deserialize normal classes for its functionality. As depicted in Figure 4, DESERIGUARD aims to construct SAPTs to record the potential complex hierarchical structures of the deserialized objects. To build SAPTs, DESERIGUARD initially identifies all deserialization entries as starting points to resolve the possible typecasting of the nested deserialized object. Subsequently, certain properties within nested objects may still lack explicit class restrictions, necessitating a recursive analysis based on the application's semantics. By inferring the types of these properties based on the application's semantics, DESERIGUARD constructs SAPTs for allowlist policy synthesis. Lastly, within this tree, DESERIGUARD gathers the possibly deserialized classes and incorporates them into the allowlist, thereby synthesizing an allowlist policy for strict deserialization restrictions.

**Semantic-Aware Property Tree.** First, we shall present the formal definition of SAPT, denoted as $\mathbb{T}$. As the right part of Figure 5 shows, SAPT is a tree that records a deserialized object's possible hierarchical structure and its properties. In detail, each node $n_x = (c_x, P_x)$ on the tree represents a class $c_x$ and its property set $P_x$. For example, the node $n_2$ is for class `Session` and has four properties. The edges $e \in E$ of SAPT fall into two types, *property edges* and *inference edges*. *Property edge* $e_p(n_x, n_y)$ connects the parent node $n_x(c_x, P_x)$ to the child node $n_y(c_y, P_y)$ when $n_x$'s property $p \in P_x$ is of class $c_y$. For instance, an edge $e_p(n_2, n_4)$ represents that the `user` property of `Session` is an instance of the `User` class. Another type of edge is the *inference edge* $e_i$, which represents the edges that are inferred from the inheritance relation and dataflow of the application. If $e_i(n_x, n_y)$ exists, the object of class $c_y$ can be assigned to the property of class $c_x$ according to the inheritance relation and dataflow of objects. Next, we will introduce the process of identifying nodes and edges to construct SAPT.

**Root Node Identification.** The root node $n_0$ of the tree corresponds to the class of the object returned by the deserialization entry, depicted as the "*Deserialization Entry*" in the left part of Figure 5. Hence, the construction of SAPT starts with locating deserialized entries of known deserialization libraries. Referring to the deserialization scanning works [24], [25], we identify the deserialization libraries that are widely used for exploitation and collected their deserialization entry methods, such as the `readObject` method of the `ObjectInputStream` object and the `fromXML` method of the XStream library. By locating the invocation of these methods, DESERIGUARD can find source nodes of our SAPT construction. DESERIGUARD does not consider the instance where attackers discover new exploitable deserialization libraries, as this is a rare occurrence and falls beyond the scope of all existing deserialization defenses [10], [16].

After obtaining the root node of SAPT, DESERIGUARD will construct SAPT by connecting the potentially deserialized classes. Usually, developers only assume the deserialized object as `Object` or `Serializable` initially [18]. Therefore, the root node of SAPT is of class `Object` or `Serializable`, as the "$n_0$: `Object`" in Figure 5 shows. However, by analyzing the existing deserialized gadgets, it is found that `Object`, `Serializable`, and `Comparable`, which we regard as *general classes*, are three common ancestors of exploitable classes. Meanwhile, they are also the ancestors of almost all classes in Java. Once general classes are added to the allowlist, DESERIGUARD may permit the deserialization of many exploitable classes. Hence, DESERIGUARD needs to find the classes actually required by the developers based on the program's semantics. To this end, from the deserialization entry, DESERIGUARD first traces the deserialized object's dataflow and looks for the typecasting that more precisely describes its class. In Figure 5, as the dataflow starting from the deserialization entry shows, a deserialized ob-

ject can be cast to `Profile` or `Session` on different control flow branches. Therefore, classes `Profile` and `Session` should be permitted at the currently analyzed deserialization entry, and nodes $n_1$ and $n_2$ are connected to the root node $n_0$.

**Property Edges Connection.** Java is an object-oriented programming language where classes have complex associations. In a Java application, a class often contains many properties (i.e., member variables) of other types, resulting in an object instance composed of numerous interconnected objects, like the `Session` class depicted in Figure 5. SAPT should record such complex structures of deserialized objects, so DESERIGUARD needs to analyze these properties recursively. To achieve this, we utilize the property edge $e_p(n_x, n_y)$ to connect the parent node $n_x$ to its property's class node $n_y$, which is depicted as the solid arrows in Figure 5. The property edges should be constructed recursively for each node on SAPT. Once a new class is added to SAPT, DESERIGUARD needs to perform the analysis on all its properties, as their types are all possible to be deserialized. In some cases, classes comprise only basic types (i.e., `String` and primitive types) and have no complex inheritance relations, allowing DESERIGUARD to terminate its analysis. For example, in Figure 5, the property `expirationTime` of node "$n_2$: `Session`" is of the class `Time`, which consists of basic types and has no subclass. Hence, DESERIGUARD can terminate the analysis on its subtree.

**Inference Edges Solvement.** Besides various properties, classes in Java applications also exhibit complex inheritance relations. When the class $c_y$ is a subclass of $c_x$, an object of $c_y$ can be assigned to the property defined with $c_x$. Hence, if $c_x$ has been constructed as a node of SAPT, we should assume that any one of its subclasses, denoted as $c_y$, may be deserialized by the application. Therefore, DESERIGUARD utilizes inference edges $e_i(n_x, n_y)$ to record such inheritance relations. For example, as illustrated by the dataflow ① in the left section of Figure 5, the class `User` has two subclasses, namely `Admin` and `Guest`. Therefore, "$n_7$: `Admin`" and "$n_8$: `Guest`" are connected as children of node "$n_4$: `User`" on SAPT with inference edges in dot lines. Once these two nodes are added to SAPT, DESERIGUARD will perform a heuristic analysis on their properties, which are all defined as basic types. In addition, these two classes have no subclass, so DESERIGUARD can terminate the analysis on two subtrees.

However, not all properties on SAPT can be analyzed as easily as the `user` property in the `Session` class. Some properties are required to be of general classes, including `Object`, `Serializable`, and `Comparable`. For instance, the property `id` of node $n_2$ belongs to the `Comparable` type, which is implemented by extensive classes. Considering the inheritance relationship, it appears that all these classes should be permitted for deserialization. Moreover, according to the definition of `Session`, node $n_2$'s property `values` is defined as `List<Object>`. To deserialize the `values` property of `Session`, it is necessary to reconstruct all the objects in the list, which implies that all subclasses of `Object` should be permitted for deserialization. Nevertheless, general classes and interfaces are extended and implemented by nearly all classes. Hence, DESERIGUARD should not indiscriminately allow the deserialization of generic classes, avoiding including potentially exploitable ones. Although developers define these properties using generic classes for convenience during development, according to the program's semantics, only a few specific classes are actually necessary and deserialized within these properties. Therefore, DESERIGUARD should infer the classes the application assumes for these properties, synthesizing a more stringent allowlist.

To accomplish this, DESERIGUARD performs dataflow tracing to deduce possible classes. The key insight behind the dataflow tracing is that objects in Java applications should be cast to the desired classes before executing their functionalities. As the dataflows ② and ③ in Figure 5 depict, the analysis of the properties with generic classes starts from the property access operations. Then, by identifying the type-related operations on the dataflow, DESERIGUARD can infer the developer's requirements of deserialized object types. The type-related operations can be divided into two categories: typecasting and type comparison. For typecasting, developers may explicitly perform typecasting or utilize `Class.cast()` method. The type comparison relies on the statements like `instanceof` and `Class.isAssignableFrom()`. For example, as shown in dataflow ③ of Figure 5, the application utilizes the `instanceof` statement to check the type of the undetermined object and subsequently executes different branches accordingly. Once the object of the general class is identified as more precise classes, DESERIGUARD adds inference edges to connect the corresponding nodes. In Figure 5, the inference edges $e_i(n_6, n_9)$, $e_i(n_6, n_{10})$, and $e_i(n_6, n_{11})$ are established according to the dataflow ③.

**Permitted Classes Identification.** Based on SAPT, DESERIGUARD can generate the allowlist policy to over-approximate all necessary classes to be deserialized. In detail, we traverse the tree and collect the classes of all nodes as the allowlist policy. However, some generic classes should not be integrated into the policy directly, like "$n_3$: `Object`" and "$n_6$: `Comparable` on the subtree of "$n_2$: `Session`". Instead, DESERIGUARD will delve into their subtrees for more strict policies. The subtrees of these generic classes' nodes are constructed according to the properties' subsequent dataflow, reflecting the possible class to which the unidentified general classes will be cast. Therefore, DESERIGUARD can deduce the actual required class based on SAPT. However, in some cases, an object's dataflow may be unresolved due to insufficient semantics in the program. To avoid missing necessary classes, DESERIGUARD will adopt the object's currently solved class as the allowlist. If the class belongs to general classes, DESERIGUARD will still include them in the allowlist, but in the meantime, alert developers that this deserialization entry lacks enough protection due to insufficient semantics.

```
"example.app.Session",   "example.app.IntID",
"example.app.User",      "example.app.AuthKey",
"example.app.HashVal",   "example.app.Time",
```

Fig. 6. The policy synthesized based on the subtree of "$n_2$: `Session`" in Figure 5. The unidentified classes are omitted.

The policy synthesized for the subtree of node "$n_2$: `Session`" is shown in Figure 6, where some unidentified classes are omitted. Since certain classes have inheritance

relations, DESERIGUARD can synthesize a more concise policy by adopting only the parent class in the allowlist policy. Therefore, if we set a rule of allowing the class `User` in Figure 5, all its subclasses are also permitted to be deserialized during policy enforcement. By constructing SAPTs based on different deserialization entries' semantics and synthesizing the allowlists on SAPTs, DESERIGUARD ultimately generates a customized stringent policy for each deserialization entry.

### B. Policy Enforcement

Following the synthesis of the policy, DESERIGUARD acquires an allowlist based on the program's semantics. Subsequently, DESERIGUARD should enforce the policy to safeguard the corresponding deserialization entry of the target application. For efficient deployment, DESERIGUARD should seamlessly enhance the application without requiring any modifications to the source code. To meet this end, DESERIGUARD utilizes a Java agent to instrument the bytecode of Java applications when each class is loaded by the Java Virtual Machine (JVM), and then performs real-time type auditing of untrusted deserialization. Moreover, developers are allowed to utilize different deserialization libraries, which should be automatically supported by DESERIGUARD. Therefore, DESERIGUARD needs to preprocess the deserialization libraries and find the proper positions for the type auditing.

**Type Auditing.** To achieve the type auditing, DESERIGUARD utilizes the Java agent to perform bytecode instrumentation on the application. This instrumentation consists of two components. First, DESERIGUARD performs the instrumentation at deserialization entries and sets a flag to enable the type auditing and indicate the policy on which the type auditing should rely. Second, for each deserialization library, DESERIGUARD finds a proper auditing position, which is located after the resolution of the object's type from the input byte stream, while before the gadget chain is triggered. With two instrumentations, DESERIGUARD can audit the object type based on the customized policies for different deserialization entries.

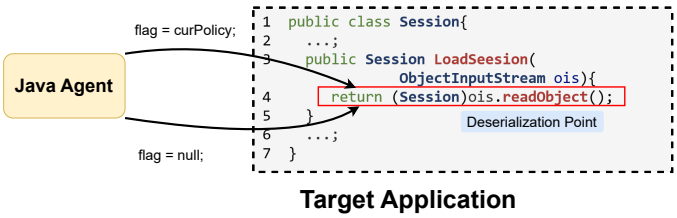


Fig. 7. Instrumentation of Java agent at the deserialization entry.

The first component of instrumentation will activate and deactivate the type auditing. Meanwhile, it will also specify the appropriate policies that the second component should enforce, since different deserialization entries need customized policies. In the first component, DESERIGUARD finds the position of each deserialization entry, which is resolved along with the deserialization policy. As illustrated in Figure 7, two invocations are instrumented, one placed directly before the deserialization entry, and another immediately after it. In these two invocations, a flag is set and unset to specify the policy for the current deserialization entry. When the flag is set, DESERIGUARD is enabled to monitor the deserialization procedure based on the customized policy. Once the deserialization is completed normally or aborted due to an unpermitted class, the flag is unset to ensure that DESERIGUARD is deactivated.

The second part of instrumentation is in the deserialization libraries adopted by the target Java application. It will perform type auditing on the class about to be deserialized following the policy indicated by the first component. As Figure 6 shows, the policy contains a series of classes, which should not be compared by name matching. Instead, DESERIGUARD will examine the class according to the inheritance relations. If the parent class is on the allowlist, all its subclasses will be permitted for deserialization.

**Auditing Position Identification.** To implement the type auditing, the Java agent needs to find a proper position to check the class of the deserialized object. To deserialize the object, all deserialization libraries first resolve the object's class using a *resolving method* and subsequently construct the object using constructor methods and reflections. During the construction, the application will invoke some specific methods that trigger the gadget chains. Therefore, DESERIGUARD should block the deserialization procedure after the target application resolves the class of the deserialized object and before the application begins to reconstruct it. However, *resolving methods* are invoked frequently in deserialization libraries. In `XStream`, we find 706 positions where *resolving methods* are invoked. Moreover, the names of *resolving methods* and the positions of their invocations vary across different deserialization libraries or even distinct versions of libraries, necessitating automatic identification of instrumentation positions. Therefore, DESERIGUARD should find the correct invocation position of the *resolving method* to perform the instrumentation.

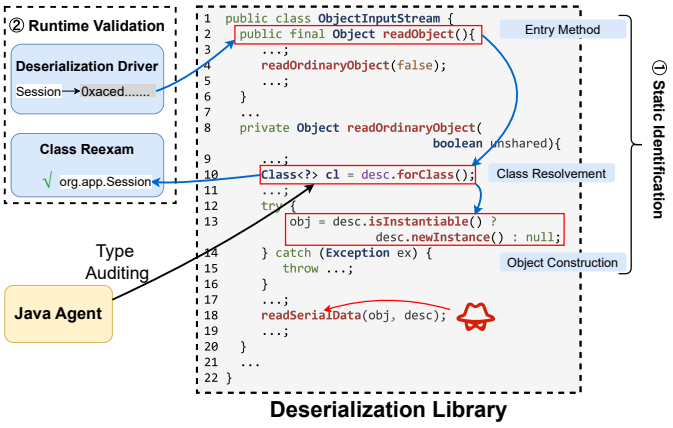


Fig. 8. Identification of Auditing Position, where the Java agent will perform the instrumentation for policy enforcement.

To meet this end, DESERIGUARD combines static identification and runtime validation to find the proper position for each library. In detail, DESERIGUARD first finds all methods that resolve `Class` object based on byte streams from libraries' exposed interfaces, like line 10 in Figure 8. Here, the `desc` object is an instance of ObjectStreamClass, which resolves the input byte stream and holds the class information of the deserialized object. Via `forClass` method, the application obtains the `Class` object, and then DESERIGUARD should determine that the library will create the corresponding

object based on the `Class` object. In this step, deserialization libraries rely on the `newInstance` method or reflection mechanism to build a constructor method of the deserialized object. Therefore, by identifying the dataflow from a `Class` object resolved based on byte stream to the construction methods utilizing the `Class` object, DESERIGUARD can identify numerous potential positions of *resolving methods*' invocations. These potential positions will undergo further examination through runtime validation.

As depicted in Figure 8, during runtime validation, DESERIGUARD first generates a deserialization driver that triggers the relevant deserialization entry point. Then, it utilizes the driver to deserialize a workload object with a specific class. For example, the driver in Figure 8 utilizes the `Session` as a workload class. Meanwhile, DESERIGUARD monitors the candidate positions to determine which one is activated, validating that the `Class` object produced by the *resolving method* corresponds to the object provided in the deserialization driver. In Figure 8, DESERIGUARD successfully gains a `Class` object for `Session`, determining the proper position for type auditing. In this way, DESERIGUARD identifies the proper invocation position of the *resolving method* and uses it as the instrumentation position for type auditing.

## V. IMPLEMENTATION

This section illustrates the implementation of DESERIGUARD. For policy synthesis, DESERIGUARD performs the analysis on the program's source code using CodeQL [24]. In detail, we utilize CodeQL to tailor the dataflow analysis, which locates the typecasting and type comparison statements on the dataflow and identifies potential types. Next, from these types, DESERIGUARD will continue to trace dataflow on their properties access statements and inheritance relations. By recursively tracing the dataflow, DESERIGUARD constructs SAPT. During the tracing, some dataflows may interfere with DESERIGUARD's analysis and need to be filtered. First, an object may be implicitly cast to its superclass. For example, an object can be sent into the method `toString(Object o)` or the map `HashMap<String, Object>`. These methods or data structures are defined with general classes, but DESERIGUARD should not identify the object as `Object`. Therefore, these implicit typecastings toward superclasses are filtered during the analysis. Moreover, when an object is pushed into a container like `HashMap`, DESERIGUARD will continually trace the dataflow of `HashMap`'s elements. However, an element of `HashMap` may be cast to various classes on different dataflow. In this case, DESERIGUARD will refer to the class of the original object and only trace the dataflows with corresponding types.

In some cases, the dataflow of an object may be hard to solve due to insufficient semantics or complex mechanisms in Java. For example, the object's subsequent dataflow involves the reflection and Java Native Interface (JNI) invocations [26], [27], which consistently pose challenges in Java analysis. To avoid missing dataflow, DESERIGUARD performs self-referencing reflection analysis to gather all potentially invoked methods that align with the calling context (i.e., type of return value and parameters) [28]. Meanwhile, DESERIGUARD utilizes CodeQL to model the frequently used JNIs and connect the dataflow. However, though mitigating the above challenges, DESERIGUARD may still fail to solve some dataflows. For instance, a deserialized object may be sent out via the network or stored on the disk. In these cases, DESERIGUARD will at least adopt the currently solved classes to over-approximate the necessary classes in the subsequent dataflow.

Moreover, for each deserialization entry in the application, DESERIGUARD will synthesize a customized allowlist policy. Although only exposed deserialization entries are exploitable and require auditing of untrusted objects, DESERIGUARD will protect all deserialization entries. It is because determining whether a deserialization entry can be accessed by attackers is hard [29]. Many vulnerabilities are caused by exposed deserialization entries that are typically thought safe. For example, in CVE-2022-40955 [30], attackers inject a malicious object into a deserialization entry through the database, which is assumed to be trusted by the developers of InLong. Furthermore, deserialization operations contribute to a minor portion of the Java applications' overall overhead, so protecting all entries will not impose heavy overhead. Therefore, DESERIGUARD will protect an application's all deserialization entries.

As for policy enforcement, we implement DESERIGUARD as a Java agent and attach it to the application during the application's initialization. In Java agent, DESERIGUARD utilizes the ASM library [31] to perform bytecode instrumentation. When the application deserializes an object, DESERIGUARD will be enabled and start to audit the object's type. In detail, DESERIGUARD loads the corresponding `Class` object and utilizes the `isAssignableFrom` method to determine whether the deserialized object is a subclass of any class in the allowlist. If one deserialized class is out of the policy, DESERIGUARD will block the deserialization process and throw an exception. To reduce overhead, DESERIGUARD caches the `Class` objects of classes in the allowlist, which saves extensive time for loading classes in repeated deserializations.

## VI. EVALUATION

In this section, we evaluate DESERIGUARD and aim to answer the following research questions:

**RQ1.** What is the performance of DESERIGUARD when defending against real-world vulnerabilities? (Section VI-A)

**RQ2.** Can DESERIGUARD synthesize more strict policies than applications' developers? (Section VI-B)

**RQ3.** How does DESERIGUARD compare to state-of-the-art approaches? (Section VI-C)

**Experiment Setup.** We first perform our experiments on 12 real-world vulnerabilities, which are collected from a survey work [18] for those with huge impact and the Common Vulnerability Exploit (CVE) for recent ones. Initially, 17 vulnerabilities are collected, among which one lacks detailed information about vulnerabilities. Moreover, four vulnerable applications do not have test cases for the deserialization entry, making it hard for us to evaluate the false alarms and overhead. Therefore, we finally reproduce 12 vulnerabilities to assess the defense performance of DESERIGUARD. To evaluate the overhead and false alarm of DESERIGUARD's defense, we execute the unit tests and integration tests of each vulnerable application. During testing, we monitor the status of the target application to capture failed deserialization and measure the

TABLE I. DESERIGUARD PERFORMANCE ON REAL-WORLD VULNERABILITIES. "NO" IN THE COLUMN "FALSE ALARMS" INDICATES NO FALSE ALARM. THE GADGET CHAINS ARE NAMED AFTER YSOSERIAL [32].

| Application | Label | LoC | Classes | Resist | Policy Rules | Permitted Classes | Gadget Chain | False Alarms |
|---|---|---|---|---|---|---|---|---|
| Apereo CAS-4.1.5 | CAS 4.1.x | 1.86M | 49.12K | ✓ | 4 | 4 | CommonsCollections4 | No |
| Richfaces-4.3.3 | CVE-2013-2165 | 57.5K | 5.74K | ✓ | 1 | 1 | CommonsCollections5 | No |
| Jenkins-1.637 | CVE-2015-8103 | 643.07K | 23.19K | ✓ | 1 | 1 | CommonsCollections3 | No |
| Shiro-1.2.4 | CVE-2016-4437 | 82.85K | 5.60K | ✓ | 79 | 79 | RMI + CommonsCollections4 | No |
| Jenkins-2.46.1 | CVE-2017-1000353 | 646.45K | 18.67K | ✓ | 28 | 161 | CommonsCollections3 | No |
| Olingo-4.6.0 | CVE-2019-17556 | 150.82K | 13.81K | ✓ | 33 | 58 | CommonsCollections5 | No |
| Tomcat-10.0.0 | CVE-2020-9484 | 171.89K | 17.50K | ✓ | 14 | 23 | Groovy1 | No |
| Ofbiz-17.02.03 | CVE-2020-9496 | 2.00M | 25.87K | ✓ | 413 | 935 | CommonsBeanutils1 | No |
| Ofbiz-17.12.05 | CVE-2021-26295 | 2.79M | 30.71K | ✓ | 623 | 1342 | RMI + CommonsBeanutils1 | No |
| Ofbiz-17.12.06 | CVE-2021-29200 | 2.09M | 27.51K | ✓ | 392 | 905 | RMI + CommonsBeanutils1 | No |
| Ofbiz-17.12.06 | CVE-2021-30128 | 2.09M | 27.51K | ✓ | 392 | 905 | CommonsBeanutils1 | No |
| Log4j-1.2.17 | CVE-2022-23307 | 695.99K | 2.91K | ✓ | 20 | 28 | CommonsCollections6 | No |

time delay in passing all tests. The evaluation is performed on an Ubuntu 20.04 with Intel i7-10700k and 48G memory. We restart the machine before each evaluation to avoid interrupts from other applications.

Moreover, we collect 109 developer-designed deserialization policies from well-developed projects on GitHub. To search for proper projects, we first identify several keywords used for configuring deserialization policies, such as " ObjectInputFilter", "extends ObjectInputStream" and "XStream addPermission". Then, by searching these keywords on all Java applications of GitHub in descending order of stars, we obtain a series of applications that may set a deserialization policy. However, since these keywords may be utilized for many functionalities, some applications may have related keywords but do not adopt deserialization policies. In addition, a complex application may utilize the deserialization mechanism at different positions, where developers may adopt different policies for distinct deserialization entries. Therefore, we consume extensive efforts to analyze each application and manually collect 109 policies formulated by developers from 40 popular applications with more than 100 stars.

### A. Defending Against Real-World Attack

We first utilize DESERIGUARD to enhance some Java applications with deserialization vulnerabilities and try to exploit them. This approach enables us to evaluate the effectiveness of DESERIGUARD in mitigating real-world deserialization attacks. Moreover, by executing comprehensive unit tests and integration tests on applications, it is demonstrated that DESERIGUARD does not impede the normal execution of applications and introduces only negligible runtime overhead.

**Defense Performance.** As depicted in Table I, on all 12 vulnerabilities, DESERIGUARD successfully resists deserialization attacks. These vulnerable applications are complicated and have 1.11M LoC on average, introducing a great challenge for developers to formulate allowlist policies. In addition, developers typically define and include an average of 20.68K classes in these applications, forming intricate inheritance relations for analysis. Among these classes, attackers may find exploitable gadget chains to implement deserialization attacks. In the "Gadget Chain" column of Table I, we depict the gadget chains used for the deserialization attack for each vulnerability. Moreover, different versions of an application may necessitate very different policies. For example, DESERIGUARD formulates an allowlist with 623 and 392 rules for Ofbiz in 17.12.05 and 17.12.06 respectively. Hence, automatic allowlist synthesis is an urgent need for frequently updated applications.

When observing the synthesized policies, it is found that on six applications, DESERIGUARD synthesizes relatively strict policies that permit less than 30 classes. This is because these vulnerable deserialization entries are specified for classes with relatively simple structures. For example, the CVE-2016-0788 exposes a deserialization entry that can only be used to deserialize `Capability`, which comprises only basic types, resulting in an allowlist with a single rule.

However, for other vulnerabilities, DESERIGUARD synthesizes more loose policies with up to 623 rules, permitting 1342 classes for deserialization. At deserialization entries originating from these applications, they may deserialize highly diverse objects, resulting in the synthesis of policies that contain numerous rules. Moreover, the property trees of deserialized objects from these deserialization entries have some generic classes (e.g., Object, Serializable), which are cast to various classes according to their dataflow. This presents a more challenging scenario where manually crafting a precise policy becomes impractical. For instance, as an enterprise resource planning framework, Ofbiz provides rich functionalities with more than 2M LoC and 25k classes. In one of these functionalities, Ofbiz should deserialize a map from the user-provided byte stream. While tracing the dataflow of maps and objects within the map, DESERIGUARD finds that this map is frequently utilized as the `context` parameter in various methods. These methods extract specific objects from the `context` and cast them to desired classes. As a result, the allowlist policy for Ofbiz should encompass all these classes, leading to a complex policy. In this case, even an experienced developer may struggle to identify all necessary classes. However, DESERIGUARD can synthesize a comprehensive allowlist policy, offering valuable support to developers.

When observing the permitted classes for the deserialization entries of these vulnerabilities, DESERIGUARD permits only a small part of classes in terms of total classes. Even with a policy comprising over 600 rules, DESERIGUARD permits only a small fraction (0.048%) of the total classes in the Ofbiz project. Within this limited subset of classes, it becomes significantly challenging for attackers to construct an exploitable gadget chain. In Section VI-C1, we will delve into

a further evaluation, demonstrating that even with an advanced gadget mining tool, attackers are unable to find usable gadgets.

**False Alarm of DESERIGUARD.** In DESERIGUARD's design, we adopt a conservative strategy in policy synthesis. Although an object's subsequent dataflow, in some cases, may not be explicitly reflected in the applications' source code. To avoid interrupting applications' normal execution, DESERIGUARD will include the object's current solved class into allowlist to over-approximate the classes that the object may be cast to. Hence, as demonstrated in the "False Alarm" column of Table I, DESERIGUARD incurs no false alarms during unit testing and integration testing provided by each application's developers. In addition, we also conduct a real-world deployment experiment on Jenkins with CVE-2015-8103 and Ofbiz with CVE-2021-26295 to evaluate the false alarm of DESERIGUARD. We collect 301,452 and 465,167 instances of deserialization with 19,586 and 20,723 distinct deserialization workloads for Jenkins and Ofbiz, respectively. During the deployment, DeseriGuard incurs no false alarm. In conclusion, DESERIGUARD can provide a strict policy without introducing false alarms on 12 complex applications.

TABLE II. OVERHEAD OF DESERIGUARD. THE COLUMN "ANALYSIS" DEPICTS THE COST OF POLICY SYNTHESIS. THE "INITIALIZATION" OVERHEAD IS INCURRED BY THE INITIALIZATION OF THE JAVA AGENT. "AUDITING" INDICATES THE TIME CONSUMPTION FOR EACH TYPE AUDITING. "SLOWDOWN" PRESENTS OVERALL OVERHEAD OF DESERIGUARD DURING APPLICATIONS' EXECUTION.

| Application | Before Runtime | | Runtime | |
|---|---|---|---|---|
| | Analysis | Initialization | Auditing | Slowdown |
| Apereo CAS-4.1.5 | 10s | 8.10ms | 0.100ms | 0.795% |
| Richfaces-4.3.3 | 6s | 2.37ms | 0.030ms | 4.296% |
| Jenkins-1.637 | 21s | 11.99ms | 0.042ms | 4.320% |
| Shiro-1.2.4 | 8s | 15.54ms | 0.032ms | 3.656% |
| Jenkins-2.46.1 | 22s | 84.71ms | 0.034ms | 2.931% |
| Olingo-4.6.0 | 46s | 3.84ms | 0.074ms | 3.201% |
| Tomcat-10.0.0 | 39s | 36.88ms | 0.031ms | 3.759% |
| Ofbiz-17.02.03 | 65s | 15.54ms | 0.017ms | 0.388% |
| Ofbiz-17.12.05 | 69s | 170.84ms | 0.024ms | 0.632% |
| Ofbiz-17.12.06 | 71s | 100.75ms | 0.021ms | 0.966% |
| Ofbiz-17.12.06 | 70s | 90.40ms | 0.020ms | 0.625% |
| Log4j-1.2.17 | 6s | 11.99ms | 0.032ms | 0.443% |
| Average | 36.1s | 46.08ms | 0.039ms | 2.168% |

**Overhead.** As illustrated by Table II, the overhead stems from three steps: static analysis, Java agent initialization, and real-time auditing. The static analysis is a preprocessing step performed only once before deployment. On average, DESERIGUARD takes 36.1 seconds to analyze an application. For all applications, DESERIGUARD typically requires less than 80s for analysis. The analysis overhead is influenced by the complexity of the application and the extent to which the deserialization mechanism is utilized within the application. In the case of Ofbiz, DESERIGUARD requires extensive tracing of dataflows to formulate over 600 rules, which contributes to longer analysis time. Notably, the static analysis only needs to be conducted once, and a longer analysis time is acceptable.

During application startup, the initialization process of the Java agent takes place, which involves DESERIGUARD loading the policies and performing bytecode instrumentation. On average, it takes 46.08 ms to complete the initialization process. As demonstrated by the "Initialization" column of Table II, this overhead roughly increases proportionally with the complexity of the policy, meaning that more intricate policies tend to result in longer initialization times for the Java agent. It is worth noting that this overhead is encountered only during the initialization phase and does not occur during regular application execution.

The most crucial aspect of overhead evaluation is the delay induced by runtime auditing, as illustrated in the "Auditing" and "Slowdown" columns of Table II. The overhead is evaluated using tests of each application. During testing, we measure the time cost for each type auditing and calculate the overall time delay for the entire testing process. Notably, deserialization operations account for a small portion of the whole runtime cost for an application. To evaluate how DESERIGUARD influences the overall efficiency of an application, we execute the test cases and measure the overall time delay caused by it. The "Auditing" column in Table II displays the average time consumption for type auditing during each deserialization operation. Across all applications, DESERIGUARD requires an average of 0.039 ms for each type auditing. As we optimize the type auditing by implementing caches of permitted classes, the average time consumption tends to decrease with increasing deserialization operations in test cases. Therefore, even with more than 600 rules in the deserialization policy, DESERIGUARD is able to complete type auditing on Ofbiz in about 0.02ms on average. When observing the impact of DESERIGUARD on the overall performance of applications, the "Slowdown" column reveals that, on average, DESERIGUARD introduces a time overhead of 2.168% across all applications. Given that each time of type auditing only introduces a negligible delay during each deserialization operation, the main factor affecting the overall overhead is the degree to which an application relies on the deserialization mechanism. In complex applications like Jenkins, there is a high dependency on the deserialization mechanism for various functionalities. As a result, it requires 4.32% more time to complete all test cases. However, in 6 of 12 cases, DESERIGUARD requires less than 1% additional time delay to pass all test cases. In summary, DESERIGUARD is demonstrated as an efficient real-time defense framework for deserialization attacks.

**Answer to RQ1**. DESERIGUARD exhibits effective defense capabilities while imposing negligible overhead and incurring no false alarms.

### B. Compared to Developer-Designed Policies

In addition to conducting experiments on real-world vulnerabilities, we also compare DESERIGUARD's automatically synthesized policies with policies designed by developers. We collect 109 protected deserialization entries from 40 well-developed applications on GitHub, each receiving over 100 stars. These applications rely on various deserialization libraries, including `XStream`, `FastJson`, `kryo`, and `ObjectInputStream`. By comparing the permitted classes at each deserialization entry, we can demonstrate that DESERIGUARD can provide more precise policies for applications.

First, we measure the compression rate of each allowlist policy, which represents the ratio of permitted classes to all classes in an application. As boxplots in Figure 10 depict, developer-designed policies typically permit more classes than

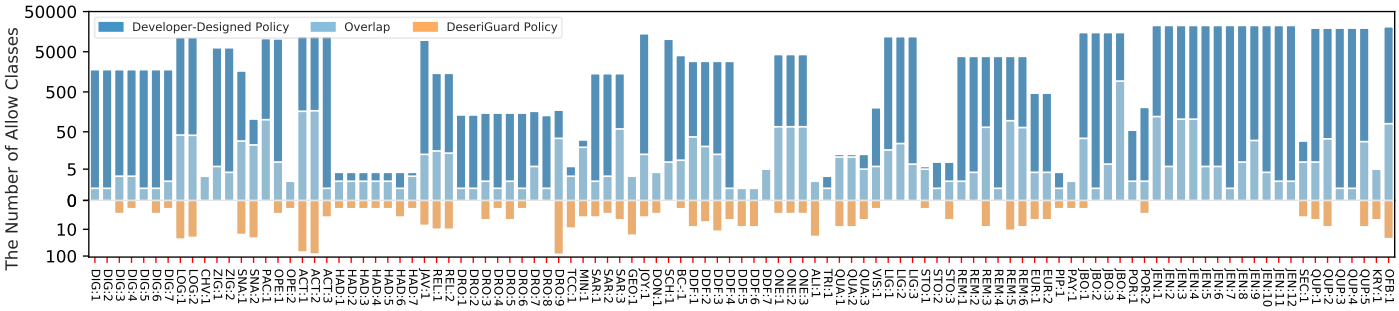


Fig. 9. The permitted classes of DESERIGUARD's policy and developer-designed policy for each deserialization entry of applications. For each bar, the light blue section represents classes that are permitted by both developers and DESERIGUARD. The deep blue and orange sections indicate classes solely allowed by developers and DESERIGUARD, respectively. Each item on the horizontal axis is the prefix of the project name and the index of the deserialization entry. Full names and deserialization entries are listed in Table VI in the Appendix.

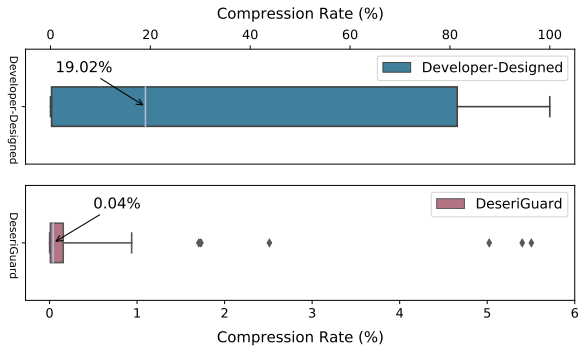


Fig. 10. Compression rate of policies from developers and DESERIGUARD.

DESERIGUARD's. In detail, DESERIGUARD's median compression rate is 0.04%, and only in 6 of 109 deserialization entries does DESERIGUARD allow the deserialization of more than 1% of an application's total classes. In the worst case, 5.504% of classes are included in the DESERIGUARD's allowlist. With such strict policies, it is hard for attackers to find exploitable gadgets in DESERIGUARD's policies, providing high security for applications. As a comparison, developer's policies achieve a 19.02% median compression rate, of which most (69 of 109) are higher than 1%. Compared to the allowlist only, on average, 0.18% and 16.28% of total classes are permitted by DESERIGUARD and developers separately, where DESERIGUARD permits 90 times fewer classes than developers. Therefore, compared to formulating policies manually, DESERIGUARD can automatically synthesize a more strict policy and save extensive manual efforts.

In most cases, DESERIGUARD can provide stricter deserialization policies. However, in 11 deserialization entries, DESERIGUARD permits 1∼19 more classes than developer-designed policies. It is because developers have formulated very strict allowlist policies for these deserialization entries, but DESERIGUARD may include some unnecessary classes. For example, on Distributed DataFrame (DDF) [33], its developers only permit the deserialization of one Class, named `CswTransactionRequest`. Meanwhile, it utilizes a *XStream* converter to customize the deserialization procedure of `CswTransactionRequest`, during which its properties are created through construction functions rather than *XStream*'s deserialization mechanism. However, during the analysis of DESERIGUARD, it needs to permit all properties of `CswTransactionRequest`, appending more rules to the allowlist. Therefore, it is a trade-off, where we permit a few more unnecessary classes to avoid restricting necessary classes. In addition, although permitting 1∼19 more classes, DESERIGUARD permits only 0.31% of total classes on average, which significantly mitigates deserialization attacks.

In addition to the compression rate, we visualize the difference of permitted classes between DESERIGUARD's policies and developer-designed policies. In Figure 9, the classes permitted by both DESERIGUARD and developers are described with light blue sections, while the deep blue sections indicate classes that are solely allowed by developers' policies. Notably, the vertical axis undergoes compression for better visualization. Despite the comparable length of sections in light blue and deep blue, developer-designed policies allow for substantially more classes than DESERIGUARD's policies. Specifically, DESERIGUARD permits 99.12% fewer classes on average. In worse cases, more than 10,000 unnecessary classes are permitted by developers' policies on applications like ActiveMQ and QuPath. The reason is that it is hard for developers to analyze the required classes precisely on complex applications. Thus, they tend to permit many classes from JDK and developer-defined classes of applications. However, after DESERIGUARD's analysis, it is found that most of these classes will not be deserialized and can be restricted.

Since DESERIGUARD aims to include all possible deserialized classes in the allowlist, it may permit the deserialization of some unnecessary classes. When observing the orange sections on the graph, we find that only several classes in DESERIGUARD's policies are excluded from developers' policies. On 39 deserialization entries, DESERIGUARD does not permit extra classes compared to the developer's policies. In the worst cases, DESERIGUARD permits 85 classes that are forbidden by ActiveMQ's developers, which accounts for only 0.26% of ActiveMQ's total classes. Overall, DESERIGUARD additional permits 0.18% of total classes on average. Therefore, DESERIGUARD can automatically synthesize an allowlist policy without introducing many unnecessary classes.

With the help of the synthesized policy from DESERIGUARD, we can avoid the situation shown in the Motivation Example (Section II-C), where a flawed deserialization policy was repeatedly bypassed on the Ofbiz project. On the one hand, DESERIGUARD can formulate a strict allowlist policy containing limited classes, making it hard for attackers to

find exploitable gadgets. On the other hand, through proper instrument and type auditing, DESERIGUARD helps developers prevent incorrect implementation of policy enforcement and provides reliable defense. In conclusion, DESERIGUARD effectively mitigates deserialization attacks and prevents potential errors that may occur during policy synthesis and enforcement.

**Answer to RQ2**. Compared to manually designed policies, policies synthesized by DESERIGUARD exhibit higher precision and strictness, thus offering enhanced levels of security.

### C. Compared to State-of-The-Art Tools

In this section, we compare DESERIGUARD with state-of-the-art approaches from two aspects, gadget mining and deserialization policy learning.

*1) Gadget Mining:* We first evaluate DESERIGUARD on a gadget mining tool called GadgetInspector [34], which can explore unknown gadget chains from classes in the application's classpath. Meantime, we evaluate DESERIGUARD on a famous gadget dataset Ysoserial [32]. By evaluating DESERIGUARD on various gadget chains, we can effectively demonstrate its ability to resist rapidly evolving gadget chains.

TABLE III. DEFENSE PERFORMANCE ON GADGET CHAINS MINED BY GADGETINSPECTOR [34] AND GADGET CHAINS FROM FAMOUS GADGET DATASET YSOSERIAL [32]. "✓" INDICATES SUCCESSFUL DEFENSE.

| Application | GadgetInspector | | Ysoserial |
|---|---|---|---|
| | Gadget Chains | Resist | Resist |
| Apereo CAS-4.1.5 | 10 | ✓ | ✓ |
| RichFaces-4.3.3 | 3 | ✓ | ✓ |
| Jenkins-1.637 | 20 | ✓ | ✓ |
| Shiro-1.2.4 | 3 | ✓ | ✓ |
| Jenkins-2.46.1 | 20 | ✓ | ✓ |
| Olingo-4.6.0 | 16 | ✓ | ✓ |
| Tomcat-10.0.0 | 15 | ✓ | ✓ |
| Ofbiz-17.02.03 | 18 | ✓ | ✓ |
| Ofbiz-17.12.05 | 19 | ✓ | ✓ |
| Ofbiz-17.12.06 | 19 | ✓ | ✓ |
| Ofbiz-17.12.06 | 19 | ✓ | ✓ |
| Log4j-1.2.17 | 2 | ✓ | ✓ |

As Table III shows, on different applications, GadgetInspector explores 2∼20 potential gadget chains. Part of these chains may still rely on some well-known gadgets, like CommonsCollections and Spring-Core, which are all included in the blocklists or removed from the allowlists by developers. However, some potential gadget chains are previously unknown, inducing significant challenges for manual policy formulation. In these cases, DESERIGUARD, with automatic policy synthesis, can successfully generate fine-grained policies and perform strict type auditing to mitigate the attacks with these potential gadgets on applications' all deserialization entries. Although not all the reported gadget chains are exploitable, DESERIGUARD can defend against all potential gadgets in case they are leveraged by attackers.

In addition to these gadget chains, we evaluate DESERIGUARD on the Ysoserial, a well-known gadget chain dataset. The results show that DESERIGUARD can block all Ysoserial's 33 gadget chains on 12 vulnerable applications. In conclusion, DESERIGUARD effectively safeguards applications against deserialization attacks by implementing strict allowlist policies with a limited number of classes. This approach significantly reduces attackers' probability of finding exploitable gadget chains, even when employing advanced gadget mining tools.

*2) Policy Learning:* Some researchers have acknowledged the challenges in Java deserialization protection and have made efforts to employ learning-based methods for resisting attacks. These approaches have yielded encouraging results and are capable of synthesizing highly stringent deserialization policies. For evaluation, we replicate the state-of-the-art defense technique known as Trusted Execution Path (Trusted) [10] and compare it with DESERIGUARD. Trusted consists of two phases, a learning phase relying on manually crafted benign deserialization workloads and a real-time policy enforcement phase based on learned policies. In the learning phase, Trusted records the benign deserialization's execution path, which reflects the order of objects to be deserialized. Then, during enforcement, it only permits deserialization fitting these known paths. To avoid the bias stemming from the crafting of benign deserialization workloads, we collect the deserialization workloads from the unit tests and integration tests of each application. After learning the trusted path on these workloads, we compare DESERIGUARD's policies with Trusted's.

First, Trusted and DESERIGUARD has different policy formats. Trusted only permits the objects whose properties' type and deserialization order exactly fit its policy, which is called path-based auditing. In contrast, DESERIGUARD adopts a type-based auditing that verifies the classes of deserialized objects without checking their deserialization order. Considering the policy format, Trusted seems to perform more strict restrictions on the deserialization procedure. However, upon analyzing the interfaces of deserialization libraries used for customizing policies, we observed that they typically focus on auditing the types of deserialized objects without considering the deserialization order. Hence, type-based auditing has been widely recognized as a highly secure approach within the community [5], [7]. Moreover, in our experiments, we achieve a 100% defense rate on all vulnerable applications, demonstrating the security of type-based auditing. Therefore, type-based auditing can effectively prevent the deserialization of classes within the gadget chain and block the attack.

TABLE IV. DEFENSE PERFORMANCE OF DESERIGUARD AND TRUSTED EXECUTION PATH.

| Applications | Defense Performance | | False Alarm | |
|---|---|---|---|---|
| | DESERIGUARD | Trusted | DESERIGUARD | Trusted |
| Apereo CAS-4.1.5 | ✓ | ✓ | No | No |
| RichFaces-4.3.3 | ✓ | ✓ | No | No |
| Jenkins-1.637 | ✓ | ✓ | No | No |
| Shiro-1.2.4 | ✓ | ✓ | No | Yes |
| Jenkins-2.46.1 | ✓ | ✓ | No | Yes |
| Olingo-4.6.0 | ✓ | ✓ | No | Yes |
| Tomcat-10.0.0 | ✓ | ✓ | No | Yes |
| Ofbiz-17.02.03 | ✓ | ✓ | No | Yes |
| Ofbiz-17.12.05 | ✓ | ✓ | No | Yes |
| Ofbiz-17.12.06 | ✓ | ✓ | No | Yes |
| Ofbiz-17.12.06 | ✓ | ✓ | No | Yes |
| Log4j-1.2.17 | ✓ | ✓ | No | No |

Table IV shows a detailed comparison of defense performance and false alarms. As for defense performance, it is

found that both Trusted and DESERIGUARD effectively mitigate all 12 vulnerabilities. It indicates that both two methods can synthesize strict allowlist policies and block malicious gadget chains. However, when observing the false alarms of the two methods' policies, it is observed that Trusted's policies may incur false alarms due to inadequent benign deserialization workloads. For example, on Tomcat, the object of the class TomcatPrincipal is possible to be deserialized, which is excluded from Trusted's policy for Tomcat. Without enough benign workloads for learning, it is hard for Trusted to traverse all possible paths of deserialization procedures, resulting in incomplete deserialization policies. In detail, after a manual analysis, we find that Trusted omits some benign classes that should be permitted for deserialization on eight vulnerabilities. We also manually craft the corresponding benign objects and send them to the unprotected applications, and these applications successfully deserialize them and perform the correct functionalities.

Collecting benign deserialization workloads poses a significant challenge for the learning-based policy synthesis approach. It is hard to manually crafting sufficient benign deserialization workloads. Therefore, DESERIGUARD is a valuable tool for developers as it enables the formulation of a strict and comprehensive allowlist policy without benign workloads.

**Answer to RQ3**. DESERIGUARD can reject the deserialization of unknown gadgets mined by advanced gadget mining tools. Meanwhile, DESERIGUARD outperforms state-of-the-art policy learning methods.

## VII. DISCUSSION

### A. Permitting Unnecessary Classes

Although DESERIGUARD has the capability to synthesize stricter policies compared to those formulated by the developers themselves, it still permits some unnecessary classes when deducing necessary classes. This decision is made to avoid interruption in the application's execution, as DESERIGUARD permits the deserialization of all classes that may be required by the application. For instance, a class may own a property that is always `null` during serialization and deserialization, but DESERIGUARD still needs to include its type in the allowlist. Though DESERIGUARD currently cannot further determine whether a class is truly necessary when it appears in SAPT, DESERIGUARD has demonstrated a 100% defense rate against 12 real-world vulnerabilities, as highlighted in Section VI-A. Moreover, DESERIGUARD also permits 99.12% fewer classes than developer-defined policies, as shown in Section VI-B. This indicates that the unnecessary classes in DESERIGUARD's policies do not undermine its effectiveness. The reason is that crafting an exploitable gadget chain from limited classes in DESERIGUARD's policies is very difficult, as blocking any classes on the gadget chain can resist the deserialization attack. In the future, we will continue to improve the analysis precision to minimize the inclusion of unnecessary classes in the allowlist policies.

### B. Straw-Man Experiment

In Section VI-A, we utilize well-known gadget chains to exploit the vulnerabilities. However, these gadget chains are widely used and have been included in the blocklist policies of many applications. It seems that these blocklist policies can also effectively mitigate deserialization attacks. To assess its effectiveness, we conduct a straw-man experiment by creating a blocklist that includes the crucial classes in Ysoserial's gadget chains before 2021, as shown by Table V in the Appendix. We then enforce it on the 12 real-world vulnerabilities and try to exploit the enhanced applications. When under the attacks of gadget chains in Table I, the blocklist is very effective and can resist all attacks. However, after analysis, we find that on Shiro-1.2.4, Apereo CAS-4.1.5, and Tomcat-10.0.0, attackers can use `AspectJWeaver` [35] gadget chain, which is found in 2021, to bypass the blocklist and perform the attack. In contrast, DESERIGUARD can effectively resist the attack of `AspectJWeaver` gadget chain on 12 applications. It indicates that blocklist policies are not strict enough and under the threat of previously unknown gadget chains.

## VIII. RELATED WORK

**Gadget Mining.** GadgetInspector [34] is the most widely used tool because it has high effectiveness and can be easily customized by developers. It utilizes the taint analysis to connect related methods and identifies chains from the deserialization entries to exploitable methods [36]. GadgetInspector has garnered increased attention from researchers in the field of gadget mining [32], [37]–[40]. After that, SerHybrid [41] is proposed for dynamic verification to reduce false positives [42]. Lai et al. [37] further improve the dynamic verification with the reflection mechanism of Java. Moreover, Tabby [43] proposes a code property graph and stores it in the neo4j database, on which developers can mine gadget chains in their projects with customized queries. Based on Tabby, GCMiner [40] further builds a Deserialization-Aware Call Graph and searches the gadget chains. Recently, ODDFuzz [16] enhances the dynamic verification step of gadget mine with fuzzing techniques. Using the guidance of seed distance and gadget coverage, it explores more combinations to craft malicious objects. Different from gadget mining tools, DESERIGUARD is a defense tool with the purpose of blocking all malicious gadget chains. Therefore, DESERIGUARD should block the deserialization of gadget chains mined by these tools.

**Deserialization Defense.** In addition to mitigating vulnerable programs through gadget mining, employing deserialization protection to safeguard applications is another important approach [44]. As attacks are completed during the object reconstruction phase of deserialization, the defense should be applied prior to it. The mainstream defense solution is known as the look-ahead defense [45], which first decodes the class information from the byte stream and audits the class. Only if the object's class is identified to be safe, will the application start to reconstruct the object. For example, JEP 290 [5] is a look-ahead defense and allows developers to customize a policy of permitting and blocking classes. Besides protection provided by JDK, many runtime application self-protection frameworks [13], [14], [46] also provide mechanisms to customize policies. They usually hook critical methods and perform auditing with byte code instrumentation [31], [47]. However, all these methods concentrate more on providing a defense mechanism with user-friendly interfaces and low overhead, ignoring the challenges involved in formulating

policies. In contrast, DESERIGUARD focuses on the automatic policy synthesis for safeguarding deserialization processes.

Some researchers have also recognized the need for automatic synthesis of deserialization policies. Trusted [10] first provides a two-phase defense framework, which first learns from benign deserialization workloads and then enforces the learned allowlist policy. However, it does not provide a promising method to produce benign workloads, resulting in insufficient learning and false alarms. François et al. [48] train a Markov chain with benign and malicious gadget chains to learn the difference between them. Then, they perform a runtime prediction on the currently deserialized object. Although it can effectively identify all malicious gadget chains, the presence of false alarms ranging from 8.44% to 11.83% hinder their practical deployment. Furthermore, due to its heavy reliance on training data, we are unable to reproduce the experimental results without its dataset. Next, Vorobyov et al. [49] introduce a learning-based approach for synthesizing a regex string filter to match the names of permitted classes. All existing methods rely on a set of benign gadgets to learn the policy, but they do not provide promising approaches to generate sufficient benign workloads, resulting in a high false alarm. Considering the potential for substantial losses caused by false alarms, DESERIGUARD is a more practical method that minimizes the occurrence of false alarms.

## IX. CONCLUSION

This paper introduces DESERIGUARD, a comprehensive approach for Java deserialization protection. By automatically formulating and enforcing policies based on application semantics, DESERIGUARD offers developers a seamless method to protect their applications. Through extensive evaluations on 12 real-world vulnerable applications, DESERIGUARD demonstrates its effectiveness by blocking all deserialization attacks. Moreover, the comparison with developer-set policies highlights DESERIGUARD's ability to restrict unnecessary classes. The performance evaluations confirm that DESERIGUARD introduces negligible overhead and does not interfere with the normal execution of policy-enhanced applications. In our future work, we aim to enhance the analysis precision of DESERIGUARD to improve its effectiveness and extend DESERIGUARD to more programming languages.

## ACKNOWLEDGMENT

This research is sponsored in part by the National Key Research and Development Project (No. 2022YFB3104000) and NSFC Program (No. 62022046, 92167101, U1911401, 62021002).

## REFERENCES

[1] J. C. S. Santos, R. A. Jones, C. Ashiogwu, and M. Mirakhorli, "Serialization-aware call graph construction," in *Proceedings of the 10th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis*, ser. SOAP 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 37–42. [Online]. Available: https://doi.org/10.1145/3460946.3464319

[2] "Cve about deserialization vulnerability," 2023. [Online]. Available: https://www.cvedetails.com/vulnerability-list/cweid-502/vulnerabilities.html

[3] "Cwe-502," 2023. [Online]. Available: https://cwe.mitre.org/data/definitions/502.html

[4] J. Dahse, N. Krein, and T. Holz, "Code reuse attacks in php: Automated pop chain generation," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 42–53. [Online]. Available: https://doi.org/10.1145/2660267.2660363

[5] OpenJDK, "Jep 290: Filter incoming serialization data," 2022. [Online]. Available: https://openjdk.org/jeps/290

[6] "Fastjson library," 2023. [Online]. Available: https://github.com/alibaba/fastjson

[7] "Xstream library," 2023. [Online]. Available: https://x-stream.github.io/

[8] "Cve-2017-1000353," 2017. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2017-1000353

[9] "Cve-2021-26295," 2021. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2021-26295

[10] 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*, M. Bailey, T. Holz, M. Stamatogiannakis, and S. Ioannidis, Eds. Cham: Springer International Publishing, 2018, pp. 445–464.

[11] OpenJdk, "Jdk project," 2023. [Online]. Available: https://openjdk.org/projects/jdk/

[12] "Jackson project," 2023. [Online]. Available: https://github.com/FasterXML/jackson

[13] Z. Yin, Z. Li, and Y. Cao, "A web application runtime application self-protection scheme against script injection attacks," in *Cloud Computing and Security: 4th International Conference, ICCCS 2018, Haikou, China, June 8-10, 2018, Revised Selected Papers, Part II 4*. Springer, 2018, pp. 566–577.

[14] "Openrasp," 2023. [Online]. Available: https://github.com/baidu/openrasp

[15] "Jenkins," 2023. [Online]. Available: https://www.jenkins.io/

[16] S. Cao, B. He, X. Sun, Y. Ouyang, C. Zhang, X. Wu, T. Su, L. Bo, B. Li, C. Ma *et al.*, "Oddfuzz: Discovering java deserialization vulnerabilities via structure-aware directed greybox fuzzing," *arXiv preprint arXiv:2304.04233*, 2023.

[17] 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," *CoRR*, vol. abs/2303.07593, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2303.07593

[18] I. Sayar, A. Bartel, E. Bodden, and Y. Le Traon, "An in-depth study of java deserialization remote-code execution exploits and vulnerabilities," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 1, feb 2023. [Online]. Available: https://doi.org/10.1145/3554732

[19] "Cve-2019-0189," 2019. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2019-0189

[20] S. Srinivasa, J. M. Pedersen, and E. Vasilomanolakis, "Deceptive directories and "vulnerable" logs: a honeypot study of the ldap and log4j attack landscape," in *2022 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, 2022, pp. 442–447.

[21] M. Sharp and A. Rountev, "Static analysis of object references in rmi-based java software," in *21st IEEE International Conference on Software Maintenance (ICSM'05)*, 2005, pp. 101–110.

[22] "Cve-2021-29200," 2021. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2021-29200

[23] "Cve-2021-30128," 2021. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2021-30128

[24] "Codeql," 2023. [Online]. Available: https://codeql.github.com/docs/

[25] C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang, "Compositional shape analysis by means of bi-abduction," *J. ACM*, vol. 58, no. 6, dec 2011. [Online]. Available: https://doi.org/10.1145/2049697.2049700

[26] Y. Li, T. Tan, and J. Xue, "Understanding and analyzing java reflection," *ACM Trans. Softw. Eng. Methodol.*, vol. 28, no. 2, feb 2019. [Online]. Available: https://doi.org/10.1145/3295739

[27] S. Liang, *The Java native interface: programmer's guide and specification*. Addison-Wesley Professional, 1999.

[28] Y. Li, T. Tan, Y. Sui, and J. Xue, "Self-inferencing reflection resolution for java," in *ECOOP 2014–Object-Oriented Programming: 28th European Conference, Uppsala, Sweden, July 28–August 1, 2014. Proceedings 28*. Springer, 2014, pp. 27–53.

[29] D. Landman, A. Serebrenik, and J. J. Vinju, "Challenges for static analysis of java reflection-literature review and empirical study," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 507–518.

[30] M. Corporation, "Cve-2022-40955," 2022. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-40955

[31] E. Bruneton, R. Lenglet, and T. Coupaye, "Asm: a code manipulation tool to implement adaptable systems," *Adaptable and extensible component systems*, vol. 30, no. 19, 2002.

[32] "Ysoserialr," 2022. [Online]. Available: https://github.com/frohoff/ysoserial

[33] "Defending against java deserialization vulnerabilities," 2023. [Online]. Available: https://github.com/ddf-project/DDF

[34] I. Haken, "Automated discovery of deserialization gadget chains," in *blackhat*, 2018.

[35] AspectJWeaver, "Aspectjweaver," 2023. [Online]. Available: https://github.com/frohoff/ysoserial/blob/2874a69f6127fd3b3f078461741910423a6b1376/src/main/java/ysoserial/payloads/AspectJWeaver.java#L46

[36] L. Sui, J. Dietrich, A. Tahir, and G. Fourtounis, "On the recall of static call graph construction in practice," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1049–1060. [Online]. Available: https://doi.org/10.1145/3377811.3380441

[37] Z. Lai, H. Qu, and L. Ying, "A composite discover method for gadget chains in java deserialization vulnerability," 2022.

[38] "Java deserialization scanner," 2021. [Online]. Available: https://github.com/federicodotta/Java-Deserialization-Scanner

[39] 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*, ser. ISSTA 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 329–340. [Online]. Available: https://doi.org/10.1145/3293882.3330576

[40] 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," *arXiv preprint arXiv:2303.07593*, 2023.

[41] 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.

[42] E. Ruf, "Context-insensitive alias analysis reconsidered," in *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, ser. PLDI '95. New York, NY, USA: Association for Computing Machinery, 1995, p. 13–22. [Online]. Available: https://doi.org/10.1145/207110.207112

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

[44] L. Carettoni, "Ddf - distributed dataframe," 2016. [Online]. Available: https://www.ikkisoft.com/stuff/Defending_against_Java_Deserialization_Vulnerabilities.pdf

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

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

[47] S. Chiba, "Javassist—a reflection-based programming wizard for java," in *Proceedings of OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, vol. 174. Citeseer, 1998, p. 21.

[48] 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*, 2022, pp. 71–75.

[49] K. Vorobyov, F. Gauthier, S. Bae, P. Krishnan, and R. O'Donoghue, "Synthesis of java deserialisation filters from examples," in *2022 IEEE 46th Annual Computers, Software, and Applications Conference (COMPSAC)*, 2022, pp. 736–745.

## APPENDIX

### A. Source Code of Figrue 5

Listing 1 exhibits the corresponding source code for Figure 5. According to the listing, the application processes each request based on a `Session` variable. If it is the user's first time sending a request, the application should create a `Session` based on the `Profile`, which is deserialized from the user's request (Lines 12∼21). If the user has made previous requests, the application can retrieve the `Session` from persistent storage through deserialization, as shown by Line 23. Then, from Lines 29 to 36, the application calculates the `priority` of the user's request, determined by the `Session`'s `id` property. The type of `id` can be one of `IntID`, `AuthKey`, and `HashVal`, each of which the application will handle accordingly. According to `Session`'s definition in Lines 41∼46, `values`, `expirationTime`, `id`, and `user` are all properties of `Session`. When adding `Session` to the SAPT, its properties should undergo recursive analysis. The property `id` is analyzed based on the dataflow in Lines 31∼36. The type of elements in `values` should be further analyzed according to the application's subsequent dataflow. The `user` is of class `User`, which owns two subclasses, `Admin` and `Guest`. The `expirationTime` is of class `Time`, which consists of all basic types. Based on the program semantics depicted in the listing, the SAPT is constructed as Figure 5 shows.

### B. Deserialization Entry of Allowlists

In Section VI-B, we collect 109 policies from 40 popular applications. Detailed information on each application's deserialization entries is listed in Table VI. In the "Application" column, we show the name of each application. In the column "Github Stars", we present the stars of these applications. Most of these applications receive over 500 stars, and all these applications receive at least 125 stars. The most popular application, Jenkins, gets 20649 stars. Therefore, these applications are widely used and well-developed, for which developers should formulate appropriate policies for their deserialization entries. The deserialization entries of each policy are listed in the "Deserialization Entry" column. Referencing the abbreviation in "Abbr." column, we could find the corresponding bar of each policy in Figure 9.

### C. BlockList of Straw-Man Experiment

In Section VII-B, we conduct a straw-man experiment by constructing a blocklist that encompasses the crucial classes used in Ysoserial's gadget chains prior to 2021. By analyzing the exploitation of each gadget chain in Ysoserial, we summarize the key classes and formulate them as a blocklist, as depicted by Table V. In the blocklist, each rule is a regex pattern that matches one class or a set of classes. This blocklist is very effective when defending against existing gadget chains. For instance, by blocking the deserialization of `ChainedTransformer`, `InvokerTransformer`, `TransformingComparator`, and `InstantiateTransformer`, all seven Ysoserial's gadget chains mined from the `CommonsCollections` library are denied for deserialization by the blocklist. However, though including all Ysoserial's gadget chains explored before 2021, attackers can still bypass the blocklist with `AspectJWeaver` gadget chain, which was found in 2021.

TABLE V. THE BLOCKLIST SYNTHESIZED BASED ON YSOSERIAL.

| | |
|---|---|
| java\.rmi\..* | com\.sun\.rowset\.JdbcRowSetImpl |
| org\.mozilla\.javascript\..* | com\.sun\.org\.apache\.xalan\.internal\.xsltc\.trax\.TemplatesImpl |
| org\.python\.core\..* | org\.jboss\.interceptor\.builder\..* |
| org\.apache\.wicket\.util\..* | org\.jboss\.interceptor\.proxy\..* |
| bsh\.Interpreter | org\.jboss\.interceptor\.reader\..* |
| bsh\.XThis | org\.jboss\.interceptor\.spi\..* |
| com\.mchange\..* | org\.springframework\.aop\.framework\.AdvisedSupport |
| javax\.sql\.ConnectionPoolDataSource | org\.jboss\.weld\.interceptor\.builder\..* |
| org\.apache\.click\.control\.Column | org\.jboss\.weld\.interceptor\.proxy\..* |
| org\.apache\.click\.control\.Table | org\.jboss\.weld\.interceptor\.reader\..* |
| clojure\.inspector\.proxy$javax\.swing\.table\.AbstractTableModel | org\.jboss\.weld\.interceptor\.spi\..* |
| clojure\.lang\.PersistentArrayMap | org\.apache\.commons\.io\.FileUtils |
| org\.hibernate\.engine\.spi\.TypedValue | org\.apache\.myfaces\.context\..* |
| org\.hibernate\.tuple\.component\.AbstractComponentTuplizer | org\.apache\.myfaces\.el\.CompositeELResolver |
| org\.hibernate\.tuple\.component\.PojoComponentTuplizer | org\.apache\.myfaces\.el\.unified\.FacesELContext |
| org\.apache\.commons\.io\..* | org\.apache\.myfaces\.view\.facelets\.el\.ValueExpressionMethodExpression |
| org\.apache\.commons\.beanutils\.BeanComparator | com\.sun\.syndication\.feed\.impl\.ObjectBean |
| org\.apache\.commons\.collections\.functors\.ChainedTransformer | org\.springframework\.beans\.factory\.ObjectFactory |
| org\.apache\.commons\.collections\.functors\.InvokerTransformer | org\.springframework\.aop\.target\.SingletonTargetSource |
| org\.apache\.commons\.collections4\.comparators\.TransformingComparator | org\.springframework\.aop\.framework\.AdvisedSupport |
| org\.apache\.commons\.collections4\.functors\.InstantiateTransformer | com\.vaadin\.data\.util\.NestedMethodProperty |
| org\.apache\.commons\.fileupload\.disk\.DiskFileItem | com\.vaadin\.data\.util\.PropertysetItem |
| org\.codehaus\.groovy\.runtime\..* | |

```
class Request{  // Request.java
    public void handleRequest(Request request){
        Object obj = null;
        Byte[] serData = PersistenceHandler.loadData(request.getCookie());
        ObjectInputStream objIn = null;
        Session session = null;
        //check if a persistent storage of Seession exists.
        if (serData == null)
            serData = request.getSerilizedProfile();
        //if a persistent Session does not exist,  load the user profile from the request.
        try{
            ObjectInputStream byteInputStream =
                    new ByteArrayInputStream(serData);
            ObjectInputStream objIn =
                    new ObjectInputStream(byteInputStream);
            obj = objIn.readObject();
        } catch (Exception e) {
            e.printStackTrace();
            return null;}
        if(obj instanceof profile){
            session = createSession((Profile) obj)
        }else{
            session = (Session)obj;}
        processSession(session);
        Object val = session.getValue(0);
        ...
    }
    public void processSession(Session session){
        Comparable userIndex = Session.id;
        int priority;
        if (userIndex instanceof IntID){
            priority = ((IntID)userIndex).toInt();
        }else if (userIndex instanceof AuthKey){
            priority = getPriority((AuthKey)userIndex);}
        else if (userIndex instanceof HashVal){
            priority = priorityMap.get((HashVal)userIndex);
        }
        ...
    }
}
class Session{  // Session.java
    private User user;
    private List<Object> values;
    private Time expirationTime;
    public Comparable id;
    public Object getValue(int index){
        return values.get(i);
    }
}
class User{   // User.java
    private String name;
}
class Admin extends User{  // Admin.java
    private String privilege;
}
class Guest extends User{  // Guest.java
    int expirationTime;
}
```

Listing 1: Source code on which the Semantic-Aware Property Tree in Figure 5 is based.

TABLE VI. INFORMATION ABOUT APPLICATIONS FOR WHICH DEVELOPERS DESIGN DESERIALIZATION POLICIES.

| No. | Application | Github Stars | Deserialization Entry | Abbr. | No. | Application | Github Stars | Deserialization Entry | Abbr. |
|---|---|---|---|---|---|---|---|---|---|
| 1 | digital | 3107 | TruthTable:47 | DIG:1 | 56 | ddf | 166 | TransactionMessageBodyReader:78 | DDF:5 |
| 2 | digital | 3107 | Circuit:147 | DIG:2 | 57 | ddf | 166 | FeatureCollectionMessageBodyReaderWfs20:202 | DDF:6 |
| 3 | digital | 3107 | CircuitTransferable:80 | DIG:3 | 58 | ddf | 166 | XStreamWfs11FeatureTransformer:74 | DDF:7 |
| 4 | digital | 3107 | FSM:96 | DIG:4 | 59 | onedev | 11064 | VersionedXmlDoc:490 | ONE:1 |
| 5 | digital | 3107 | SettingsBase:46 | DIG:5 | 60 | onedev | 11064 | VersionedXmlDoc:493 | ONE:2 |
| 6 | digital | 3107 | Configuration:74 | DIG:6 | 61 | onedev | 11064 | VersionedXmlDoc:500 | ONE:3 |
| 7 | digital | 3107 | Bundle:48 | DIG:7 | 62 | Alink | 3366 | DLPredictServiceMapper:243 | ALI:1 |
| 8 | logback | 2715 | SocketNode:63 | LOG:1 | 63 | triplea | 936 | GameDataComponent:43 | TRI:1 |
| 9 | logback | 2715 | SocketNode:83 | LOG:2 | 64 | quasar | 4516 | KryoSerializer:119 | QUA:1 |
| 10 | Chvote | 707 | SafeObjectReader:58 | CHV:1 | 65 | quasar | 4516 | ReplaceableObjectKryo:112 | QUA:2 |
| 11 | zigbee4java | 138 | ZigBeeNetworkStateSerializer:44 | ZIG:1 | 66 | quasar | 4516 | CollectionsSetFromMapSerializer:61 | QUA:3 |
| 12 | zigbee4java | 138 | NetworkStateSerializer:76 | ZIG:2 | 67 | visicut | 212 | FilebasedManager:301 | VIS:1 |
| 13 | snap-engine | 166 | GraphIO:83 | SNA:1 | 68 | light-task-scheduler | 2983 | BeanUtils:27 | LIG:1 |
| 14 | snap-engine | 166 | ModuleManifestParser:59 | SNA:2 | 69 | light-task-scheduler | 2983 | FastJSONAdapter:23 | LIG:2 |
| 15 | pac4j | 2287 | JavaSerializer:65 | PAC:1 | 70 | light-task-scheduler | 2983 | JavaSerializable:42 | LIG:3 |
| 16 | opengrok | 3899 | Definitions:321 | OPE:1 | 71 | storm | 8880 | DefaultStateSerializer:97 | STO:1 |
| 17 | opengrok | 3899 | Scopes:179 | OPE:2 | 72 | storm | 8880 | KryoValuesDeserializer:42 | STO:2 |
| 18 | activemq | 2159 | XStreamWireFormat:65 | ACT:1 | 73 | storm | 8880 | WindowKryoSerializer:67 | STO:3 |
| 19 | activemq | 2159 | XStreamWireFormat:70 | ACT:2 | 74 | remoting | 205 | Capability:185 | REM:1 |
| 20 | activemq | 2159 | SubQueueSelectorCacheBroker:201 | ACT:3 | 75 | remoting | 205 | ClassLoaderHolder:40 | REM:2 |
| 21 | hadoop | 13507 | IOStatisticsSnapshot:270 | HAD:1 | 76 | remoting | 205 | Command:155 | REM:3 |
| 22 | hadoop | 13507 | IOStatisticsSnapshot:272 | HAD:2 | 77 | remoting | 205 | RemoteInputStream:187 | REM:4 |
| 23 | hadoop | 13507 | IOStatisticsSnapshot:274 | HAD:3 | 78 | remoting | 205 | UserRequest:289 | REM:5 |
| 24 | hadoop | 13507 | IOStatisticsSnapshot:276 | HAD:4 | 79 | remoting | 205 | TrafficAnalyzer:26 | REM:6 |
| 25 | hadoop | 13507 | IOStatisticsSnapshot:278 | HAD:5 | 80 | eureka | 11779 | CodecWrappers:387 | EUR:1 |
| 26 | hadoop | 13507 | IOStatisticAssertions:517 | HAD:6 | 81 | eureka | 11779 | CodecWrappers:349 | EUR:2 |
| 27 | hadoop | 13507 | ZKConfigurationStore:319 | HAD:7 | 82 | pippo | 777 | SerializationSessionDataTranscoder:52 | PIP:1 |
| 28 | javamelody | 2842 | CounterStorage:157 | JAV:1 | 83 | payara | 856 | OpenTracingIiopServerInterceptor:109 | PAY:1 |
| 29 | reload4j | 133 | SocketNode:78 | REL:1 | 84 | jboot | 717 | JsonBodyParseInterceptor:55 | JBO:1 |
| 30 | reload4j | 133 | LoggingReceiver:70 | REL:2 | 85 | jboot | 717 | ApiDocUtil:200 | JBO:2 |
| 31 | drools | 5299 | XmlBifParser:58 | DRO:1 | 86 | jboot | 717 | FastJsonSerializer:56 | JBO:3 |
| 32 | drools | 5299 | XmlBifParser:70 | DRO:2 | 87 | jboot | 717 | JsonUtil:320 | JBO:4 |
| 33 | drools | 5299 | KieModuleMarshaller:88 | DRO:3 | 88 | portfolio | 2280 | ClientFactory:122 | POR:1 |
| 34 | drools | 5299 | KieModuleMarshaller:93 | DRO:4 | 89 | portfolio | 2280 | ECBExchangeRateProvider:91 | POR:2 |
| 35 | drools | 5299 | KieModuleMarshaller:98 | DRO:5 | 90 | jenkins | 20649 | XmlFile:165 | JEN:1 |
| 36 | drools | 5299 | KieModuleMarshaller:103 | DRO:6 | 91 | jenkins | 20649 | CreateNodeCommand:55 | JEN:2 |
| 37 | drools | 5299 | ScenarioSimulationXMLPersistence:202 | DRO:7 | 92 | jenkins | 20649 | XStream2:230 | JEN:3 |
| 38 | drools | 5299 | Jenerator:101 | DRO:8 | 93 | jenkins | 20649 | XStream2:233 | JEN:4 |
| 39 | drools | 5299 | XStreamMarshaller:151 | DRO:9 | 94 | jenkins | 20649 | Computer:1558 | JEN:5 |
| 40 | tcc-transaction | 5616 | KryoPoolSerializer:94 | TCC:1 | 95 | jenkins | 20649 | ComputerSet:266 | JEN:6 |
| 41 | mina-sshd | 715 | SimpleGeneratorHostKeyProvider:65 | MIN:1 | 96 | jenkins | 20649 | XmlFile:196 | JEN:7 |
| 42 | saros | 155 | XMPPAccountStore:165 | SAR:1 | 97 | jenkins | 20649 | View:1418 | JEN:8 |
| 43 | saros | 155 | ColorIDSetStorage:178 | SAR:2 | 98 | jenkins | 20649 | StreamTaskListener:196 | JEN:9 |
| 44 | saros | 155 | XStreamExtensionProvider:296 | SAR:3 | 99 | jenkins | 20649 | CloudSet:207 | JEN:10 |
| 45 | geoserver | 3240 | DefaultTileLayerCatalog:469 | GEO:1 | 100 | jenkins | 20649 | InstallUtil:292 | JEN:11 |
| 46 | joyrpc | 412 | KryoReader:48 | JOY:1 | 101 | jenkins | 20649 | XStreamDOM:170 | JEN:12 |
| 47 | DongTai-agent-java | 617 | SerializeUtils:61 | DON:1 | 102 | security | 125 | Base64Helper:180 | SEC:1 |
| 48 | schemacrawler | 1424 | JavaSerializedCatalog:48 | SCH:1 | 103 | qupath | 809 | QuPathGUI:2134 | QUP:1 |
| 49 | log4j2 | 3102 | SortedArrayStringMap:622 | LOG:1 | 104 | qupath | 809 | PathIO:353 | QUP:2 |
| 50 | log4j2 | 3102 | OpenHashStringMap:734 | LOG:2 | 105 | qupath | 809 | PathIO:196 | QUP:3 |
| 51 | bc-java | 1973 | XMSSUtil:331 | BC-:1 | 106 | qupath | 809 | PathIO:218 | QUP:4 |
| 52 | ddf | 166 | CswRecordConverter:189 | DDF:1 | 107 | qupath | 809 | PathIO:698 | QUP:5 |
| 53 | ddf | 166 | MetacardImpl:870 | DDF:2 | 108 | kryonet | 1775 | KryoSerialization:73 | KRY:1 |
| 54 | ddf | 166 | GetRecordsMessageBodyReader:159 | DDF:3 | 109 | ofbiz | 562 | UtilObject:96 | OFB:1 |
| 55 | ddf | 166 | GmdTransformer:242 | DDF:4 | | | | | |