Crystallizer: A Hybrid Path Analysis Framework to Aid in Uncovering Deserialization Vulnerabilities

Prashast Srivastava
Purdue University
United States

Flavio Toffalini
EPFL
Switzerland

Kostyantyn Vorobyov
Oracle Labs
Australia

François Gauthier
Oracle Labs
Australia

Antonio Bianchi
Purdue University
United States

Mathias Payer
EPFL
Switzerland

ABSTRACT

Applications use serialization and deserialization to exchange data. Serialization allows developers to exchange messages or perform remote method invocation in distributed applications. However, the application logic itself is responsible for security. Adversaries may abuse bugs in the deserialization logic to forcibly invoke attacker-controlled methods by crafting malicious bytestreams (payloads).

Crystallizer presents a novel hybrid framework to automatically uncover deserialization vulnerabilities by combining static and dynamic analyses. Our intuition is to first over-approximate possible payloads through static analysis (to constrain the search space). Then, we use dynamic analysis to instantiate concrete payloads as a proof-of-concept of a vulnerability (giving the analyst concrete examples of possible attacks). Our proof-of-concept focuses on Java deserialization as the imminent domain of such attacks.

We evaluate our prototype on seven popular Java libraries against state-of-the-art frameworks for uncovering gadget chains. In contrast to existing tools, we uncovered 41 previously unknown exploitable chains. Furthermore, we show the real-world security impact of Crystallizer by using it to synthesize gadget chains to mount RCE and DoS attacks on three popular Java applications. We have responsibly disclosed all newly discovered vulnerabilities.

CCS CONCEPTS

• **Security and privacy → Software and application security**.

KEYWORDS

Deserialization vulnerabiltiies, Java, hybrid analysis

ACM Reference Format:
Prashast Srivastava, Flavio Toffalini, Kostyantyn Vorobyov, François Gauthier, Antonio Bianchi, and Mathias Payer. 2023. Crystallizer: A Hybrid Path Analysis Framework to Aid in Uncovering Deserialization Vulnerabilities. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23), December 3–9, 2023, San Francisco, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3611643.3616313

ESEC/FSE '23, December 3–9, 2023, San Francisco, CA, USA

ACM ISBN 979-8-4007-0327-0/23/12.
https://doi.org/10.1145/3611643.3616313

1 INTRODUCTION

Serialization is a key feature in modern languages (*e.g.*, Java, C#, or PHP) that enables cross-platform communication, remote method invocations, and object persistence. Serialization converts object graphs into bytestreams. Symmetrically to serialization, deserialization rebuilds the original object graph from the bytestream. By default, deserialization ensures that the deserialized objects are valid but it does not enforce security constraints. Security (both during and after deserialization) is the sole responsibility of the application logic. Incomplete security checks allow attackers to bend the control-flow/data-flow of a program. These attacks can hijack the deserialization process, granting the attacker remote code execution (RCE), denial of service (DoS), or information persistence capabilities such as Arbitrary File Writes (AFW). Deserialization vulnerabilities have shown catastrophic security impact [24]. *E.g.*, the Equifax data breach [16] was caused by a deserialization vulnerability enabling RCE in the [36]. More recently, the `Log4Shell` vulnerability in the widely used `Log4j2` library can be exploited in newer versions of the JDK that were previously thought safe by leveraging deserialization-based attack vectors [28].

Payloads for deserialization attacks are composed of nested objects that, when deserialized, force the application to invoke an attacker-controlled sequence of methods, also called a *gadget* chain. The last gadget of the chain is usually called *sink* and may invoke system functions, *e.g.*, `Runtime.exec()` with attacker-specified arguments, allowing the attacker to execute arbitrary system commands. The gadgets in the deserialization domain are conceptually similar to gadgets in Return-Oriented Programming (ROP) [48, 51] for binary exploitation: small pieces of code in the vulnerable program that are stitched together by an attacker. However, deserialization gadgets do not operate at the machine code level, instead, they bend the serialization logic to express malicious actions.

Attack chains heavily depend on the application logic. Therefore, finding such gadget combinations that bypass the application logic is crucial to fix vulnerabilities. As of now, discovering deserialization vulnerabilities is predominantly manual and requires solving three main challenges:

C1. Sink Gadgets Identification: New sink gadgets that are useful to the attacker are currently identified through heuristics, *e.g.*, marking calls to `Runtime.exec()`. However, we observe this approach overlooks non-trivial sinks and inhibits discovering other interesting types of attacks (*e.g.*, DoS).

C2. Large State Space: The search space for gadget chains in current applications is massive with thousands of gadget combinations. This makes finding a gadget chain that can be used to mount a deserialization attack is akin to finding a needle in a haystack.
C3. Complex Payload Creation: Deserialization payloads require careful instantiation of classes and arguments that obey the execution constraints of the gadget chain. Consequently, valid bytestream creation becomes exceedingly complex due to the large number of possible combinations that nested objects can assume.

To overcome the aforementioned challenges, we design Crystallizer: a hybrid framework that combines static and dynamic analysis to synthesize concrete payloads for gadget chains and find deserialization vulnerabilities automatically. First, our framework identifies new sink gadgets in an application. Then, it uses static analysis to construct a *gadget graph*: a data structure that encodes all possible gadget chains within a target software (up to a certain length). This greatly reduces the explorable state space for gadget chains. Crystallizer creates payloads as bytestreams out of the reduced state space dynamically. Our framework synthesizes payloads in a chain-aware manner: it keeps track of the execution chain order and performs a best-effort approach to create well-formed arguments for each of the gadgets while obeying language semantics. We implement our proof-of-concept tool for Java as it is widely adopted as the backbone for software development [9, 50].

We evaluate Crystallizer on seven libraries and three applications. Across the seven libraries, it finds 41 new chains in addition to seven previously known gadget chains [22]. This demonstrates Crystallizer's ability to find both existing and new gadget chains automatically. Furthermore, we compare Crystallizer against two state-of-the-art tools [26, 47] for finding Java-based deserialization vulnerabilities and showcase that Crystallizer drastically outperforms existing state-of-the-art in terms of finding exploitable gadget chains. Finally, we showcase the real-world security impact of Crystallizer by synthesizing payloads that we use to demonstrate DoS and RCE attacks on three popular Java applications. The corresponding proof-of-concept exploits were responsibly disclosed. In summary, this paper makes the following contributions:

- We perform a systematic analysis of how deserialization vulnerabilities manifest themselves in the form of gadget chains, including challenges to uncover them automatically.
- We present Crystallizer, a hybrid framework to automatically uncover deserialization vulnerabilities by crafting payloads that exercise gadget chains in the target.
- We evaluate it against seven libraries and find 41 new chains in addition to seven previously known chains.
- Crystallizer outperforms state-of-the-art tools for finding Java-based deserialization vulnerabilities and demonstrate real-world security impact by using it to mount DoS and RCE attacks on three popular real-world applications.
- All our evaluation artifacts along with the source code of our framework are made available at https://github.com/HexHive/Crystallizer.

2 DESERIALIZATION ATTACKS

We discuss the basics of Java serialization. Then, we establish terminology relevant to deserialization attacks and showcase an example attack on a popular Java-based library *Apache Commons Collections* [1]. Finally, we discuss domain-specific challenges.

2.1 Serialization and Deserialization

Serialization is the action of transforming objects into a bytestream. Deserialization later rebuilds the objects from the received stream. Serialization for Java employs the `Serializable` interface [38]. Serialized objects of classes that implement this interface can be created using the `writeObject` method provided by the JDK [40]. The method encodes the object's fields into a bytestream to, *e.g.*, send it across the network or store it into a file. On the other end, the method `readObject` [39] deserializes the byte stream and rebuilds the original object automatically. Note that the deserialized object's class must be in the `classpath` [42], otherwise deserialization fails. Java allows specifying custom serialization and deserialization routines to instruct the receiver application about custom data processing, *i.e.,* post-processing data while filling an object's fields. As these mechanisms allow great flexibility, they also leave a large exploitable attack surface.

2.2 Payload Formalization

Let us establish terminology relevant to deserialization attacks. A gadget is any invoked method during deserialization. It forms the basic building block for an attack. A gadget chain corresponds to a sequence of method invocations triggered upon deserialization of a *payload*. *Payload* refers to a bytestream corresponding to a set of serialized nested objects. A payload that exploits a deserialization vulnerability forces the application to call an attacker-specified gadget chain which can be used to mount an attack, *e.g.,* RCE. In general, a deserialization attack is possible because the deserialization process automatically rebuilds the received object from the attacker-specified bytestream and, in doing so, potentially enables attacker-specified code to be executed.

Gadgets fall into three categories [35]: (i) **Trigger Gadgets** are the first elements invoked during deserialization and serve as the attack's entry points. In Java, such gadgets are usually classes that override specific magic methods (*e.g.,* `readObject()`). Custom deserialization routines operate on data that may be attacker-controlled allowing the trigger gadgets to kickstart a chain, (ii) **Link Gadgets** orchestrate the flow of attacker-controlled data from a trigger to a sink gadget, and (iii) **Sink Gadgets** launch the attack by running attacker-specified malicious actions.

Our **Gadget Graph** represents an over-approximation of all the possible *gadgets chains* in a program. Hence, a payload exercises only a specific path in the graph between the trigger gadget and the sink gadget. Since gadgets are the methods executed through the standard deserialization process, we model the gadget graph as a subcomponent of the application callgraph whose nodes are marked as gadgets (trigger, link, or sink). §3.1 describes our approach to extracting the gadget graph.

2.3 Payload Example

We present a known deserialization attack on `Apache Commons Collections` library explaining: (i) execution flow of a gadget chain vulnerable to a deserialization attack, and (ii) the creation of a payload that exercises this vulnerable chain.

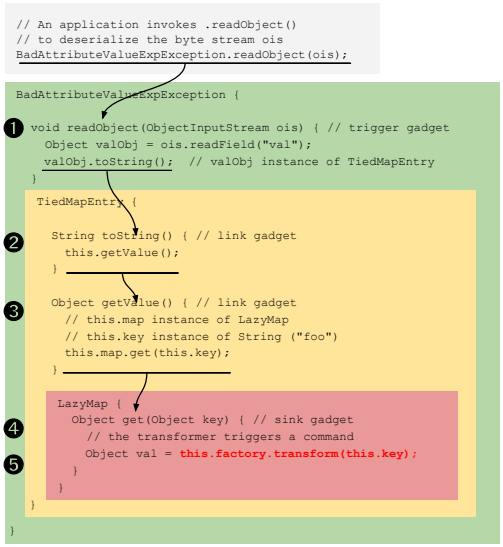

Figure 1: A simplified example for the gadget chain executed upon the payload (Listing 1) being deserialized.

Figure 1 shows the vulnerable gadget chain. The `readObject()` (❶) method of the `BadAttributeValueExpException` class is executed first, making it the trigger gadget. This gadget rebuilds the object (instance of `BadAttributeValueExpException`) from the bytestream and invokes a `toString()` method on one of its field members (`val`). The object `valObj` is an instance of the class `TiedMapEntry`, then its `toString()` method is called (❷) which in turn calls its `getValue()` method. The `getValue()` method retries a key from a map (❸). If the map is an instance of `LazyMap`, it will try to build an item corresponding to the key parameter "`foo`" (❹) by using a `Transformer` class whose object can be instantiated in such a way that the item building performs RCE (❺). Since executing the gadget `get()` method inside the `LazyMap` can lead to RCE [32], we categorize it as a sink gadget. The gadgets belonging to `TiedMapEntry` are referred to as link gadgets since they chain the invocation from the trigger gadget to the sink gadget.

The gadget chain highlights two key observations: (i) the gadget chain is a subgraph of the application callgraph, and (ii) exercising this gadget chain requires a bytestream that is crafted from a set of nested objects in such a way that the above gadget chain is invoked.

The payload that exercises the above-mentioned gadget chain to achieve RCE is shown in Listing 1. The first step is to instantiate a `Transformer` that executes `exec("/bin/bash")` (Line 8). The `Transformer` is then used to instantiate a `LazyMap` object (Line 12). The `LazyMap` automatically instantiates any missing entry using the `Transformer` class instance; thus invoking `exec()`. We then use the `LazyMap` to build a `TiedMapEntry` (Line 14) and bind it to a `BadAttributeValueExpException` instance (Line 17). Specifically, this class overrides the `readObject()` method and acts as our trigger gadget. `val` is the final payload which is serialized (Line 21) to a bytestream, ready to be sent to a vulnerable application.

```
1  // command to execute
2  final String[] execArgs = { "/bin/bash" };
3
4  // Preparing object for Transformer which
5  // is used inside the sink gadget to grant RCE to an attacker
6  final Transformer[] transformers = new Transformer[] {
7      new InvokerTransformer("exec", new Class[]
8      { String.class }, execArgs), /*...*/ };
9
10 final Map innerMap = new HashMap();
11 // Preparing object for LazyMap which acts as the sink gadget
12 final Map lazyMap  = new LazyMap(innerMap, transformers);
13 // Prepraring object corresponding to a link gadget
14 TiedMapEntry entry = new TiedMapEntry(lazyMap, "foo");
15
16 // Preparing object corresponding to the trigger gadget
17 BadAttributeValueExpException val = new BadAttributeValueExpException(val);
18
19 ObjectOutputStream os = new ObjectOutputStream(new
   ↪ FileOutputStream("payload.bin"));
20 // Writing the object into serialized bytestream (payload)
21 os.writeObject(val);
```

Listing 1: Simplified Java code creating the payload targeting *Apache Commons Collection*. Figure 1 describes the observed control flow execution upon deserializing this payload.

2.4 Challenges

Recalling the example in Listing 1, we identify three main challenges for automating chain creation: sink gadget identification, large state space, and complex payload generation.

C1 Sink Gadgets Identification. While trigger gadgets are easy to locate (*i.e.,* they are overrides of known magic methods such as `readObject()`), link gadgets are generic nodes in a gadget graph. Identifying sink gadgets requires non-trivial code knowledge. Previous works use heuristics [26] to locate the usage of specific functions (*e.g.,* `Runtime.exec()`). However, we observe that they overlook a large group of alternate sinks. Therefore, we adopt a broader definition: a gadget is considered a sink if it may operate on objects of *any* type. We identify such gadgets by checking if they may use objects of type `Object` [41]. Since `Object` corresponds to the root of the class object hierarchy, a gadget operating on this type can operate on objects of any class. We chose this definition because (i) it may result in a higher chance of manipulating the gadget to perform attacker-specified functionality, and (ii) it allows CRYSTALLIZER to target and find a wide spectrum of threats (*e.g.,* logic-based DoS chains) that were missed by previous works.

C2 Large State Space. To estimate the explorable state space of gadget chains, we conduct a preliminary analysis in *Apache Commons Collections*. First, we extract a callgraph through Soot [52] and then build a gadget graph on top of it (see §3.1). The callgraph consists of 2,009 gadgets and 38,579 edges. Our analysis reduces this large space to 295 gadgets and 2,168 edges in our gadget graph.

Even within a gadget graph, the number of candidate chains to be explored is still large, thus necessitating automated exploration. We quantified candidate gadget chains in this gadget graph from trigger to sink gadgets using a Djikstra-like algorithm [23]. To keep the analysis concise, we upper-bound the maximum length of discovered candidate chains. For a maximum path length of 5, there are 25,866 candidate chains to be explored.

C3 Complex Payload Generation. Payloads are composed of well-formed objects that obey the execution constraints of the gadget chain. In Listing 1, a `LazyMap` object requires instances of `Map`

and `Transformer` to be passed to its constructor (Line 14). Moreover, we need to obey the language semantics and pass objects as arguments that implement the respective `Map` and `Transformer` interfaces. Next, as we create the object for the predecessor gadget (`TiedMap`), we must ensure that the previously created object for `LazyMap` is correctly passed as an argument. Therefore, building concrete payloads that exercise gadget chains is challenging because it requires: (i) inference of correct parameters and (ii) instantiation of valid connections between objects.

3 CRYSTALLIZER DESIGN

Crystallizer is a hybrid path analysis framework to automatically uncover deserialization vulnerabilities by finding gadget chains in targets. Given a gadget graph, our intuition is to automatically identify the sink gadgets and then find possible paths leading to sinks that can be instantiated as a set of connected objects (§2.3).

Crystallizer produces payloads as long as there exists a sequence of gadgets that reach a sink. Crystallizer takes information about trigger gadgets and a target as input, then it outputs concrete payloads that execute the gadget chain, demonstrating potentially exploitable gadget chains. Developers can use this information to patch deserialization bugs; attackers can use adjust the parameters to fine-tune the execution of the chain. Figure 2 shows an overview of Crystallizer's three components: Static Analysis Module (❶ in Figure 2), Sink Identification (❷ in Figure 2), and the Probabilistic Concretization phase (❸ in Figure 2).

3.1 Static Analysis Module

This module takes a library and information about trigger gadgets as input and produces a gadget graph. The information provided about trigger gadgets is in the form of methods invoked by a trigger gadget. Crystallizer uses this information to automatically infer which methods in a library can be used as entry points. Looking at our example in §2.3, all `toString` methods present in the target library are treated as entry points into the target library. Leveraging this abstracted view of the trigger gadgets is in line with prior works for automated discovery of deserialization attacks [11, 47].

We build the gadget graph in four steps. (i) We extract an over-approximated callgraph using Class Hierarchy Analysis (CHA) [18] from the target software using the entry points described above, (ii) In the callgraph, we select all classes that implement the `Serializable` interface directly or through one of their ancestors (§2.1), and mark all their methods as gadgets, (iii) We use the trigger gadget information to mark the entry points in the gadget graph accordingly, while we mark all the other nodes as link gadgets, and (iv) Finally, we discard all nodes that are unreachable from trigger gadgets. The gadget graph produced by this module has only the entry points and link gadgets marked, while we mark the sinks in this gadget graph with the help of the Sink Identification module.

3.2 Sink Identification

Starting from the Static Analysis Module's gadget graph, we infer which gadgets can be used as sinks. Here, we describe the sink definition in §2.4: gadgets that use arbitrary class objects. Our module enables Crystallizer to identify sinks for RCE, DoS, or AFW. We finally mark the sink gadgets in the gadget graph accordingly.

To infer sink gadgets, Crystallizer performs a two-step process. First, it dynamically infers candidate gadgets that may use arbitrary objects. Second, a set of static filters validates if the candidate gadgets use arbitrary objects. The candidate gadgets not filtered out are flagged as sink gadgets. The dynamic inference gives initial evidence of whether a gadget may perform malicious actions and the static inference incorporates access patterns to increase precision.

Dynamic Inference. Crystallizer flags gadgets that may use an arbitrary object either as one of its declaring classes' fields or as a method parameter passed to the gadget itself. It performs this dynamic inference with the help of a *honeypot* class—a serializable class that raises an exception when instantiated. Crystallizer randomly picks one of the reachable gadgets from the gadget graph and flags it as a candidate for static filtering if it can instantiate an object of the honeypot class into (i) one of the field members of the declaring class, or (ii) one of the method parameters can be instantiated with the honeypot class. Crystallizer flags a candidate gadget, if one of the previous two conditions is fulfilled. Crystallizer also logs the argument type through which the honeypot class was instantiated (referred to as the *tainted* argument type). This information is used during the static filtering phase for making Crystallizer more precise in identifying sink gadgets.

Static Filtering. The flagged candidates must pass a set of static filters. These static filters are necessary to weed out gadgets that do not use *tainted* arguments. The filters are based on the characteristics of known sinks. We use the argument type instead of the actual argument through which the honeypot class was instantiated for filtering since there can exist multiple arguments (field members or method parameters) of the same type. If a candidate gadget passes through any of the static filters then it is flagged as a sink gadget. In case a field member was used to load in the honeypot class, we apply a set of three filters: (i) We flag gadgets that directly refer to a field having the same type as the tainted argument. (ii) We extend the previous analysis to all reachable methods using a field with the same tainted argument type. (iii) We also flag gadgets that indirectly use the tainted argument. We identify indirect usage by checking if the tainted argument is cast to another type in the class constructor and then see usage for this new type in the gadget. However, in case the argument is loaded in through a method parameter then we flag the gadget if any of the method parameters corresponding to the tainted argument were used in a method invocation.

3.3 Probabilistic Concretization

Leveraging the gadget graph, we propose a probabilistic method to generate payloads that trigger deserialization vulnerabilities. We achieve this goal by using three modules. First, we use a Candidate Chain Extractor module to find a gadget chain that connects a trigger and a sink. Second, we feed the candidate chains to a Dynamic Analysis Module, which attempts to create a payload for the corresponding chains. Finally, we submit the payload to the Deserialization Probing module that deserializes the payload and returns feedback to the Dynamic Analysis Module. The feedback can be adjusted according to the threat model and recognize chains exhibiting the intended behavior. Specifically, we show how adopting different heuristics enables us to identify RCE, AFW, or DoS

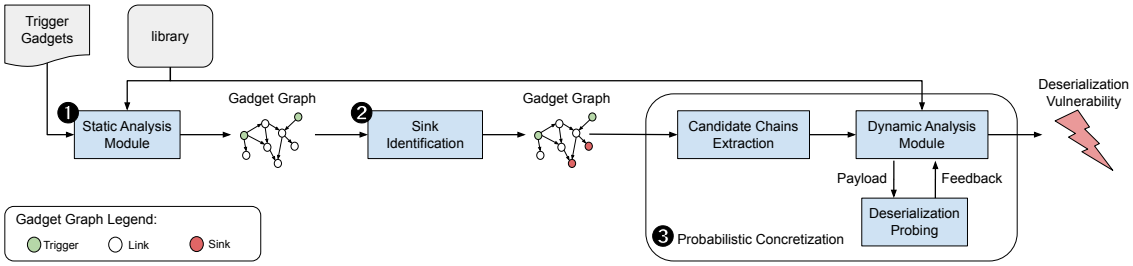

Figure 2: Architectural overview of CRYSTALLIZER.

chains. CRYSTALLIZER adopts a dynamic approach to concretization to ensure that it only reports chains for which it can create payloads that exercise them. This is in stark contrast to purely static approaches that are plagued with false positives, *i.e.,* reporting chains that cannot really be exercised due to not taking into consideration the execution constraints of the chain or the language semantics (discussed in §5.3)

Candidate Chain Extractor. This module uses a Djikstra-like algorithm [23] to identify candidate gadget chains that map paths from entry points to sink gadgets. We further define a threshold to upper-bound the length of candidate chains. Without this threshold, the state space of candidate gadget chains would become intractable for an exhaustive exploration. In our experiments, we set a threshold of up to five gadgets as inspired by known exploitable gadget chains.

Dynamic Analysis Module. A gadget chain is fully concretized if there exists an input payload that exercises the gadget chain when passed to a deserialization entry point. To concretize a gadget chain, we instantiate objects for each of the gadgets in the chain. The objects must provide two prerequisites: language-specific (in our case Java): create well-formed for declaring classes of the gadgets, and chain-specific: instantiate the objects in such a way that the execution flows successfully from one gadget to another.

Based on the insight described above, we present our concretization methodology for gadget chains in Algorithm 1. The procedure takes as input a candidate gadget chain and outputs a payload that can be tested by the Deserialization Probing module. The concretization process instantiates the nodes in reverse order, *i.e.,* from sink to target (Line 4). We adopt this strategy to fulfill the chain-specific prerequisite described previously. Furthermore, this allows the algorithm to terminate early if no objects can be instantiated.

To satisfy chain-specific prerequisites, CRYSTALLIZER uses an *object cache* to store previously instantiated objects. When a node is passed to the `ObjectFactory` for instantiation (Line 11), it checks if the object cache contains an object of the same type, or can be cast into, the requested node. If these conditions are met, we distinguish two cases. (i) The object has the same type as the requested node. Thus, we reuse it as is (Line 15). (ii) The object can be cast into the requested node type. Thus, we randomly create a new object or return the existing one from the cache (Line 17). We perform this action randomly instead of in a guided manner since reasoning about the semantics is more expensive than just exercising all possible combinations. If the object cache does not contain suitable objects, then we instantiate a new node (Line 20) by satisfying

Algorithm 1 Dynamic Analysis Module

```
Input: Candidate Gadget Chain G
Output: Payload for concretized gadget chain P
 1: procedure ConcretizeChain(G)
 2:     objectCache ← ∅
 3:     revIterator ← G.nodes.revIterator()
 4:     while revIterator.hasPrevious() do
 5:         node ← revIterator.previous()
 6:         object ← ObjectFactory(node, objectCache)
 7:         objectCache.put(object)
 8:     end while
 9:     return P ← objectCache.getTopNode()
10: end procedure
11: procedure ObjectFactory(node, objectCache)
12:     cls ← node.getDeclaringClass()
13:     if objectCache.isPreInstantiated(node) then
14:         if objectCache.hasExactType(node) then
15:             mustReturn(objectCache, node)
16:         else
17:             mayReturn(objectCache, node)
18:         end if
19:     end if
20:     clsObj ← cls.pickConstructor().instantiate();
21:     return clsObj
22: end procedure
```

the language-specific prerequisites. For primitive data types, we use a pre-defined finite set created from commonly-used values in known vulnerabilities sourced from Ysoserial [22]. For user-defined data types, we instantiate using a randomly chosen constructor synthesizing the required parameters recursively if necessary.

Deserialization Probing. Once a payload is successfully instantiated, we submit it to the Deserialization Probing to test if the payload expresses the intended behavior, *i.e.,* RCE, AFW, or DoS. We use different feedback according to the attack we detect. For RCE and AFW, CRYSTALLIZER reports a payload if it can execute each gadget in the chain from the trigger to the sink. To track gadget chain execution, we use method-level coverage feedback. However, to transform the chain in a concrete exploit, human assistance is needed to fine-tune the concretized payload (discussed in §5.1). For the DoS chains, instead, we are interested in payloads that keep the CPU busy for a long time, therefore, we consider the deserialization execution time as feedback. Specifically, we consider possible DoS chains that require time more than a given threshold to be executed (5s in our experiments). In contrast to RCE/AFW payloads, no human intervention is needed since the synthesized payload by itself exhibits the intended behavior.

We, on purpose, use the same sink gadgets for RCE, AFW and DoS chains. Our intuition is that a sink operating on arbitrary classes can be easily tuned to express different attacks by combining heuristics and different feedback.

4 IMPLEMENTATION

Here, we describe the static analyzer, Dazzer—our Probabilistic Concretization tool built on top of Jazzer [12], and the method-level instrumentation.

Static analyzer. We develop our static analyzer on top of Soot version 4.2.1 [44]. Soot is the standard tool for analyzing Java bytecode and provides built-in analysis for callgraph and class hierarchy [18]. Our analyzer consists of 1.1K Java LoC.

Dazzer. To assist the object creation, we develop Dazzer. Our tool aids the payload synthesis in Dynamic Analysis Module (§3.3) and the identification of sink gadgets in Sink Identification (§3.2). Dazzer extends Jazzer, which is originally designed to fuzz methods in isolation by creating concrete arguments for them. In contrast, Dazzer is designed to perform effective gadget chain concretization which requires adopting unique and generalized strategies for object creation. We devise the three strategies based on our analysis of numerous previously known deserialization-based vulnerabilities and deriving commonalities in terms of how they manifest themselves. First, we make the object creation chain aware by introducing the concept of a probabilistic object cache. Second, in addition to regular instantiation, Dazzer employs reflection-based strategies to force object creation if no public constructors are available, which we employ during payload creation of a gadget chain. Finally, we improve the capabilities of the object creation module to handle the generation of "generic objects" of type `Object` [41]. Jazzer only returns null objects when requested objects of type `Object`. In contrast, Dazzer not only returns commonly-used objects in chain executions such as strings and hashmaps but more importantly extends it to use the object cache which was instrumental in helping Crystallizer to concretize gadget chains. Overall, we added 2K Java LoC on top of the original Jazzer.

Method-level Feedback. Crystallizer creates an instrumented version of the target library by adding method-level coverage feedback at the bytecode level. We use Soot to insert instrumentation at the start of each method to log its execution. We use this feedback during Probabilistic Concretization for identifying concretized gadget chains(§3.3). The method-level feedback and deserialization tracing support were implemented in 470 Java LoC.

5 EVALUATION

Our evaluation of Crystallizer revolves around *five* research questions.

RQ1: Can Crystallizer find deserialization vulnerabilities in previously well-tested libraries? (§5.1)
RQ2: How does Crystallizer perform against state-of-the-art tools? (§5.2)
RQ3: How do Crystallizer's components influence the gadget chain discovery? (§5.3)
RQ4: What sinks does Crystallizer find? (§5.4)
RQ5: Can Crystallizer detect novel deserialization vulnerabilities in enterprise software? (§5.5)

Environment. We evaluate Crystallizer on seven popular Java-based libraries (Table 1) and three popular enterprise applications (§5.5). These cover a diverse range of functionality and have been previously well-tested for deserialization vulnerabilities. Moreover, we compare Crystallizer against two related tools: Gadget Inspector [26] and Rasheed et al. tool [47]. We evaluate on an Intel Xeon E5-2450 2.1GHz processor with 47G RAM running Ubuntu 20.04. Crystallizer is configured to be run in single-threaded mode and was compiled with `javac` version 11.0.11.

Table 1: Evaluation Benchmarks paired with their ground truth chains.

Benchmark	Version(s)	Description	GT Vuln
Apache Commons Collections (ACC 3.1)	3.1	Data Structure Manipulation	[32]
Apache Commons Collections (ACC 4.0)	4.0	Data Structure Manipulation	[33]
Aspectjweaver	1.9.2	Language Feature Extension	[27]
Beanshell	2.05b	Embeddable interpreter	[34]
Beanutils	1.9.2	Utility Library	[20]
Groovy	2.3.9	Object-oriented Language	[21]
Vaadin	7.7.14	Web Application Development	[31]

5.1 RQ1: Library-based evaluation

We assess the effectiveness of Crystallizer at uncovering deserialization vulnerabilities by running it on the previously well-tested seven libraries described in Table 1. To run Crystallizer on these libraries, we follow the methodology in Figure 2.

First, Crystallizer creates gadget graphs as a part of the Static Analysis Module. We provide information about a known trigger gadget (sourced from Ysoserial [22]) for each of the libraries to Crystallizer (§3.1). Crystallizer employs four unique methods (`toString`, `compare`, `hashCode`, `invoke`) to automatically identify entry points into the library. Table 2 details the size of the graphs for each target library as well as the time taken to create them along with the number of entry points used. After the gadget graph is created, we perform Sink Identification for which we allocate a time budget of one hour since it is a dynamic process.

In the Probabilistic Concretization phase, Crystallizer identifies candidate gadget chains and then attempts to concretize them. We allocate a time budget of up to 24 hours for this phase. Table 3 provides an overview of this phase. Across all seven libraries, Crystallizer concretizes 837 gadget chains. We manually deemed 604 chains as being *interesting*, i.e., the sink gadgets in these chains perform semantic functionality that could be potentially exploitable. From these 604 chains, 48 were manually validated to be exploitable.

The sink gadgets in interesting chains perform a wide range of potentially exploitable semantic functionality. Certain sink gadgets perform traditionally vulnerable functionality like using reflection to invoke arbitrary methods or writing arbitrary bytestreams to files. However, there is also a subset of sinks that are performing functionality that would not be categorized as traditionally vulnerable but when coupled with other primitives provided by the target, they become exploitable. A representative example of such a sink is `LazyMap.get()` (shown in Figure 1). This sink gadget allows using classes called `Transformers` that allow transformations to be performed on the key that is being inserted into the map. It is possible to use a set of `Transformers` which when executed mount an RCE attack. Crystallizer owing to its Sink Identification can identify not only the `LazyMap.get()` method but also all `Transformers` that are instrumental in mounting the RCE attack.

27.84% of the concretized chains are not deemed interesting since the sinks do not perform exploitable functionality. This included functionality such as wrapping objects into containers like hashmaps. These sinks are flagged because our current methodology for Sink Identification only infers whether a sink gadget may operate on potentially attacker-controlled objects but does not reason about the semantic functionality performed on such objects. We plan to integrate this semantic functionality reasoning as a part of future work to make our Sink Identification more precise.

To assess the exploitability of the gadget chains concretized by CRYSTALLIZER, we manually see if the payload for a concretized gadget chain showcasing a potential deserialization vulnerability that can be tweaked to mount an exploit. The exploitability is assessed with the help of a synthetic application that deserializes user-provided data and has the vulnerable library on its application classpath. This methodology is in line with the approach adopted by Park et al. [46] to perform their library-based evaluation. Using the methodology outlined above, we confirm exploitablity of 48 chains concretized by CRYSTALLIZER by successfully mounting RCE attacks for six out of the seven libraries and an Arbitrary File Write attack for the remaining library (`Aspectjweaver`).

The amount of manual effort required to convert a payload synthesized by CRYSTALLIZER into a working payload varies. The payloads synthesized for `Vaadin`, `Beanutils`, and `ACC4.0` by CRYSTALLIZER did not require any further manual tweaking to mount an exploit. For `Aspectjweaver` and `Groovy`, we perform minimal tweaking where only the `String` parameters used in the sink gadget are adjusted to mount the exploit. The remaining two libraries, `ACC3.1` and `Beanshell` require additional reasoning about the library semantics to convert the synthesized payload by CRYSTALLIZER into a payload that mounts an exploit. Specifically, we have to infer what primitives provided by the library could be used as parameters in the sink gadget to call `exec()` with an attacker-controlled string. §6 provides a detailed discussion of manual effort.

Finally, we perform a deeper analysis of the chains that are concretized by CRYSTALLIZER. The first observation is that CRYSTALLIZER successfully discovers the seven known ground truth chains (listed in Table 1) across **all** our evaluation targets. In addition to finding these ground truth chains, CRYSTALLIZER concretizes new gadget chains as well. Figure 3 shows the time taken by CRYSTALLIZER to create payloads for exploitable gadget chains.

Table 4 summarizes our findings with respect to the novel chains uncovered: CRYSTALLIZER automatically concretizes up to 17 previously undiscovered chains per library, that are composed of up to *six* gadgets. We quantify the complexity of the novel chains by measuring the unique classes they are composed of. Intuitively, the more unique instantiated classes a chain contains, the more language and chain-specific prerequisites CRYSTALLIZER fulfills (§3.3). Our results show the novel chains are more complex than the ground truth ones, containing twice as many unique classes. We present an example of a novel gadget chain in Listing 2. As demonstrated, through its automated reasoning about gadget chains, CRYSTALLIZER uncovers gadget chains corresponding to complex paths.

Takeaway: CRYSTALLIZER can both synthesize payloads for previously known chains in libraries, as well as create concrete payloads for novel gadget chains in well-tested libraries in an efficient manner.

Table 2: Gadget graph size of the target libraries and the time taken by CRYSTALLIZER to create it along with the number of entry points used to create the graph.

Benchmark	# Entry Points	Gadget Graph #gadgets	Gadget Graph #edges	Time (s)
ACC 3.1	41	295	2,168	73
ACC 4.0	12	573	4,069	40
Aspectjweaver	174	440	3,108	112
Beanshell	8	357	1,882	86
Beanutils	13	73	490	80
Groovy	1,170	110	271	113
Vaadin	34	2,119	8,378	153
Average	207	567	2,909	94

Table 3: Candidate chains explored by CRYSTALLIZER along with chains that were successfully concretized, chains that were deemed to be interesting, and chains that were manually validated to be exploitable.

Benchmark	Gadget Chains			
	Candidates	Concretized	Interesting	Confirmed Exploitable
ACC 3.1	25,866	689	479	7
ACC 4.0	2,23,367	4	4	4
Aspectjweaver	794	74	74	17
Beanshell	915	6	4	1
Beanutils	629	32	32	16
Groovy	1,146	7	3	1
Vaadin	31,095	25	8	2
Average	40,544	120	86	7

Table 4: Novel gadget chains found by CRYSTALLIZER along with their average gadget frequency and a comparison of the unique classes present in the discovered ground truth chain and the novel chains.

Benchmark	#Novel Chains	Avg Gadgets	Unique Classes #Known	Unique Classes #Novel
ACC 3.1	6	5	2	4
ACC 4.0	3	4	2	4
Aspectjweaver	16	6	3	5
Beanshell	0	—	1	—
Beanutils	15	4	1	3
Groovy	0	—	1	—
Vaadin	1	3	2	3
Average	**6**	**4**	2	**4**

```
// trigger
BadAttributeValueExpException.readObject();
 // links
 TiedMapEntry.toString();
 TiedMapEntry.getValue();
  SingletonMap.get();
  SingletonMap.isEqualKey();
   FastArrayList.equals();
    // sink
    LazyMap.get();
```

Listing 2: A simplified example of a gadget chain corresponding to a novel path found by CRYSTALLIZER.

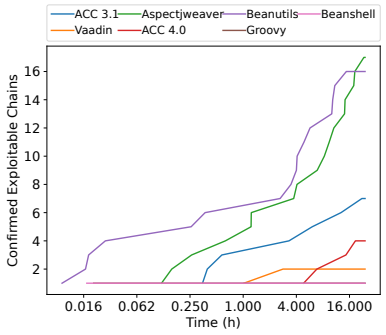

Figure 3: Time required by Crystallizer to discover the exploitable gadget chains.

5.2 RQ2: Comparison against state-of-the-art tools

We compare Crystallizer against two state-of-the-art tools for finding Java-based deserialization vulnerabilities:
(i) **Gadget Inspector** [26] is a pure static analysis tool that, given a library as input, uses a set of heuristics to report potential gadget chains. This tool does not create concrete payloads.
(ii) **Rasheed et al.** [47] employ heap abstractions [29] to identify gadget chains corresponding to deserialization-based vulnerabilities. This tool creates concrete payloads, similar to Crystallizer.

Crystallizer v/s Gadget Inspector. We compare Crystallizer against Gadget Inspector by running both tools on the library dataset specified in Table 1 and evaluate the reported gadget chains. For this experiment, we run Crystallizer end-to-end on the libraries (illustrated in Figure 2). Furthermore, we configure both tools to uncover gadget chains corresponding to attack patterns that have been previously found in these libraries (RCE in all libraries except for `aspectjweaver`, in which an Arbitrary File Write (AFW) exists). The reason behind this configuration is two-fold. First, this configuration ensures feature parity with Gadget Inspector, since the latter cannot detect DoS chains like Crystallizer. Second, it allows us to use known chains from available datasets [22] to validate false negatives, *i.e.,* exploitable chains that exist but are undiscovered. For Crystallizer, we execute the Sink Identification for 1-hour and Probabilistic Concretization for 24 hours. Gadget Inspector terminates in a few minutes.

Table 5 shows our finding. Crystallizer uses its hybrid analysis methodology to find confirmed exploitable chains for mounting the targeted attack in *all* libraries in our dataset. Specifically, Crystallizer finds previously known exploitable chains in addition to previously unknown ones. Conversely, Gadget Inspector discovers only one exploitable chain for the `ACC 3.1` library and misses the previously known exploitable chains in the remaining six libraries.

We investigate the exploration methodology adopted by Gadget Inspector to understand why it does not find the previously known exploitable chains. One of the reasons was that, as a part of its exploration methodology, once it deems a gadget as *explored* based on its set of employed heuristics, it does not try to uncover any chains further using the same gadget. This strategy prevents Gadget Inspector from reporting certain gadget chains. We find a concrete example of this in `Vaadin`. This shows the importance of exercising and exploring alternative paths while performing gadget chain discovery, as done by Crystallizer.

Table 5: Comparison of Gadget Inspector against Crystallizer in terms of gadget chains reported for libraries and the ones which were confirmed to be exploitable.

Benchmark	Gadget Inspector		Crystallizer	
	Reported	Exploitable	Concretized	Exploitable
ACC 3.1	2	1	689	7
ACC 4.0	3	0	4	4
Aspectjweaver	3	0	74	17
Beanshell	0	0	4	1
Beanutils	0	0	32	16
Groovy	2	0	7	1
Vaadin	3	0	25	2
Average	1.9	0.1	**120**	**6.9**

We investigate if we can create exploitable payloads for any of the chains reported by Gadget Inspector. First, three chains reported by Gadget Inspector in three out of the 7 libraries (`ACC 3.1`, `Aspectjweaver`, and `ACC 4.0`) are not exploitable due to incorrect reasoning about Java language semantics. For example, in some chains, Gadget Inspector incorrectly assumes that class members declared as `transient` [8] are attacker-controlled. Second, since Gadget Inspector is a static tool, it does not give any guarantees about whether it is possible to create a concrete payload. This drastically inhibited the ability to build exploitable payloads for the remaining eight out of the 13 reported chains. As an example, all the three reported chains in `Vaadin` use a gadget that required an HTTP servlet session to be setup upon instantiation and hence was beyond the scope of our assessment since the chain relied on external factors. In contrast, Crystallizer does not face such issues since the dynamic approach of Crystallizer ensures a chain is paired with concrete payloads.

Crystallizer v/s Rasheed et al. Here, we compare Crystallizer against the results presented in the paper by Rasheed et al. Ideally, we would perform a comparative evaluation similar to Gadget Inspector, but were unable to do so. Specifically, it failed while running the path analysis algorithm on our evaluation dataset.

Consequently, we compare against their reported results for `ACC 3.1` and `ACC 4.0` since these are the only two libraries in their dataset for which they were able to create a concrete payload.

For each of these libraries, their tool only found one path corresponding to a known ground truth chain for which they manually created a concrete payload. In contrast, Crystallizer not only concretized payloads to the two ground truth chains but also nine new gadget chains (shown in Table 5). This drastic performance difference can be attributed to our hybrid analysis methodology. Instead of relying on heavyweight value-flow analysis to build heap access paths, which can be prone to imprecision, our use of lightweight static analysis to build the gadget graph coupled with our dynamic analysis module that performs path concretization allows us to uncover and concretize more gadget chains.

Takeaway: Crystallizer is more effective at uncovering and creating concrete payloads for gadget chains than the existing state-of-the-art tools using its hybrid analysis methodology.

Table 6: Comparison of Crystallizer against Crystallizer-NG in terms of gadget chains reported for libraries and the ones which were confirmed to be exploitable.

Benchmark	Crystallizer-NG		Crystallizer	
	Reported	Exploitable	Concretized	Exploitable
ACC 3.1	4	0	689	7
ACC 4.0	0	0	4	4
Aspectjweaver	0	0	74	17
Beanshell	6	1	6	1
Beanutils	1	1	32	16
Groovy	9	1	7	1
Vaadin	20	0	25	2
Average	5.7	0.4	**120**	**6.9**

5.3 RQ3: Comparative Performance Evaluation

Since Crystallizer employs a hybrid path analysis methodology, we evaluate the relative importance of its static and dynamic components. We create a variant of Crystallizer that attempts to synthesize concrete payloads for a gadget chain without a gadget graph (Crystallizer-NG). However, we equip Crystallizer-NG with the knowledge of trigger gadgets and serializable gadgets to create a stronger baseline for comparison. Given this knowledge, Crystallizer-NG uses the same Probabilistic Concretization module as used in Crystallizer and attempts to uncover exploitable gadget chains by creating concrete payloads for them.

This approach is an appropriate evaluation candidate since turning off any of the other components would create variants that have a weaker capability set: (i) disabling Sink Identification would create a variant that marks *all* gadgets as sinks leading to a path explosion problem making the results meaningless, (ii) replacing our path concretization module with a vanilla fuzzer would also be weaker since it would not know how to generate objects. By comparing Crystallizer against Crystallizer-NG, we can get an accurate estimate of the benefits of building a gadget graph and using it to uncover gadget chains. Similar to our evaluation of Crystallizer, we deploy Crystallizer-NG on seven target libraries for 24 hours.

Table 6 presents an overview of the results. First, Crystallizer is 21.1x and 17.3x more performant on average than Crystallizer-NG in concretizing gadget chains and uncovering exploitable chains respectively. Second, as evident, the three exploitable gadget chains that Crystallizer-NG uncovers are in three libraries (`Beanshell`, `Beanutils`, and `Groovy`) each of which are (i) previously known, and (ii) simplest to construct requiring only one class to be instantiated correctly. In addition to previously known ones, Crystallizer can uncover novel gadget chains that are exploitable and drastically more complex (as shown previously in Table 4).

Takeaway: With the help of a gadget graph, Crystallizer reduces the state space that it explores creating 21.1X more concrete payloads for gadget chains and finding 17.3X more exploitable ones.

5.4 RQ4: Sink Identification Evaluation

We perform an in-depth analysis of the sinks detected with our framework as a part of the library-based evaluation(§5.1). We also evaluate the efficacy of the static filters used by Crystallizer at improving the precision of Sink Identification (discussed in §3.2).

Table 7: "Pre-filtering" refers to the set of sink gadgets flagged by Sink Identification's oracle. "Post-filtering" shows the number of remaining sink gadgets after applying the static filters. These are the sinks that Crystallizer tries to concretize paths to. "% reduction" refers to the difference between the number of pre- and post-filtered sinks.

Benchmark	Pre-filtering (Sinks)	Post-filtering (Sinks)	% reduction (Sinks)
ACC 3.1	403	148	63.3
ACC 4.0	647	221	65.8
Aspectjweaver	72	11	84.7
Beanshell	116	83	28.4
Beanutils	44	5	88.6
Groovy	152	36	76.3
Vaadin	681	326	52.1
Average	302	119	65.6

```
1  // previous gadgets
2  ...
3  // sink
4  FastArrayList.equals();
5   // JDK method
6   java.util.AbstractMap.equals();
7    // link
8    LazyMap.get();
```

Listing 3: A simplified chain showing how an exploitable payload was created by creating a route through a JDK function.

We detect two new sinks in `ACC 3.1` that led to six new exploitable chains missed by Gadget Inspector. For one of the exploitable chains, Crystallizer marked `FastArrayList.equals()` as a sink and created a concrete payload to reach this sink from a trigger gadget. Upon tinkering with this payload, we noticed that if `FastArrayList` were to be instantiated with a `LazyMap`, we manually found a way to exercise known dangerous functionality (`factory.transform`) by routing it through a JDK function (`AbstractMap.equals`) as shown in Listing 3. This particular chain was not reported by Gadget Inspector, because according to its analysis, it did not infer that `FastArrayList.equals()` could be routed to dangerous functionality which as we showed is not the case. This example shows our approach can find non-trivial sinks.

Filters are useful when performing sink identification We evaluate the effectiveness of static filters in making the Sink Identification more precise. Specifically, the filters ensure the tainted arguments that can be attacker-controlled are used by the gadget under consideration (discussed in § 3.2). Precision while performing Sink Identification is important since it directly impacts the number of gadget chains explored. The results of this evaluation are presented in Table 7. We see that the filtering is highly effective in drastically reducing the state space to be explored by removing 66% of the sinks that are not using the tainted argument.

Takeaway: The Sink Identification is suitable for discovering non-trivial sink gadgets and the static filters it employs are effective at filtering false positive candidate sink gadgets.

5.5 RQ5: Crystallizer in-the-wild

To showcase the effectiveness of Crystallizer at finding deserialization vulnerabilities in the wild, we deploy it on two popular

```
// trigger
BadAttributeValueExpException.readObject();
 // links
 TiedMapEntry.toString();
 TiedMapEntry.getValue();
  LazyMap.get();
   ClosureTransformer.transform();
    // sink
    WhileClosure.execute();
     // links
     TruePredicate.evaluate();
     NOPClosure.execute();
```

Listing 4: Gadget chain showcasing DoS behavior.

Apache applications: `Pulsar` [6] and `Kafka` [4] and mount two novel attacks. Specifically, we mount a RCE attack against `Pulsar` and a DoS attack against `Kafka`. These vulnerabilities are responsibly disclosed and acknowledged by the maintainers. Furthermore, to show generalizability, Crystallizer rediscovers a previously known RCE vulnerability (CVE-2020-2555) [37] in a vulnerable version of the Oracle Coherence library [43].

Kafka. Kafka is a framework that enables building data processing pipelines [7]. It provides the ability to capture data from varying sources which in turn can then be stored and processed. Kafka uses entities called *connectors* that move data in and out of Kafka as serialized bytestreams [14]. Consequently, the deserialization of untrusted data that may be attacker-controlled opens up Kafka to attacks mounted using deserialization-based vulnerabilities.

Kafka uses Java-based serialization and deserialization to store and retrieve data from a file on a local file system. Since the file that it uses for storage could be manipulated by an attacker, it employs a filtering-based mechanism to prevent deserialization of a set of specific classes [2]. The primary insight we had from the denylist is that it did not prevent deserialization of *all* classes belonging to known gadget chains but only classes that were instrumental in mounting known attacks for RCE specifically.

Based on the above insight, we deploy Crystallizer to synthesize gadget chains to mount DoS attacks instead. Crystallizer found a chain in the `Apache Commons Collections` library that exhibits DoS behavior. Specifically, Crystallizer synthesized a chain that upon deserialization performs the semantic action of executing an infinite loop (`while(1)`). The gadgets employed in the chain are shown in Listing 4. Evidently, none of the gadgets used in the chain are a part of the denylist employed by Kafka. This in turn allowed us to mount a DoS attack on the latest release of Kafka (as of February 2023) with the help of this chain.

Pulsar. Pulsar provides a framework for server-to-server messaging. As a part of its messaging subsystem, it provides extended functionality using light-weight processes to process messages. These compute processes allow for employing Java-based serialization and deserialization for message handling [3]. Processing messages that point to untrusted data makes Pulsar prone to deserialization attacks. There is no serialization filtering performed by the deserialization API used by Pulsar [5]. Therefore, it is possible to mount a deserialization-based attack using any of the classes present in the application's classpath. For Pulsar (v2.2.0), we noticed that the classpath includes the Commons Collections library.

```
// trigger
BadAttributeValueExpException.readObject();
 // links
 LimitFilter.toString();
 ChainedExtractor.extract();
    // sink
    ReflectionExtractor.extract();
```

Listing 5: Vulnerable gadget chain in Coherence uncovered and concretized by Crystallizer

Crystallizer discovered a gadget chain in this library with which we mounted an RCE attack against Pulsar.

Coherence. Coherence is an in-memory data storage that allows fast access to key-value data. It is integrated as part of Oracle Weblogic which is a popular application server. Weblogic interfaces with user-provided data, so a vulnerability found in this library allows mounting an attack through the Weblogic server.

Owing to the large size and the underlying complexity of the Coherence library (13M), the initial gadget graph constructed by Crystallizer is large containing `19,734` gadgets and `143,357` edges. Crystallizer then runs the sink identification phase over this gadget graph for one hour. At the end of this phase, it identifies 57 potential sinks and 103,598 candidate gadget chains for concretization. From these candidate chains, it concretizes 19 unique chains over five days across 20 campaigns. From these 19 unique chains, one is manually validated to be a previously known vulnerability in the Coherence library (CVE-2020-2555) [37]. In addition, seven of these chains are confirmed to be alternative paths to the same vulnerable sink including more complex paths as well. Finally, the remaining 11 are paths concretized to three unique sinks that we deemed as not performing interesting semantic functionality.

The chain concretized by Crystallizer is presented in Listing 5. Crystallizer identifies `ReflectionExtractor.extract` as a sink since it has a reference to an array of type `java.lang.Object` which is instantiated with our honeypot class during its declaring class instantiation. The payload which Crystallizer constructs to concretize the candidate chain is presented in Listing 6. Crystallizer by use of its chain concretization strategy augmented with an object cache (Algorithm 1) enables it to concretize this chain without manual adjustment. Specifically, while invoking the setter methods for `LimitFilter`, instead of generating a new object, it retrieves an object (`cObj`) from its object cache (`cObj`). Additionally, Crystallizer did not instantiate a `ReflectionExtractor` object explicitly since it inferred how to build it indirectly by instantiating a `ChainedExtractor` object with a `String`.

Takeaway: Crystallizer effectively leverages the complete application classpath to launch attacks against real-world enterprise applications even in the presence of specific bypass protections.

6 DISCUSSION

The manual effort required to analyze concretized chains by Crystallizer is lower than expected. The reason is that we can reuse knowledge across chains in the form of the unique sinks that they target. For `Aspectjweaver` (§ 5.1), instead of analyzing 74 concretized chains, we only had to examine 2 sinks manually. This strategy works because the exploitability of a concretized gadget chain hinges on whether the sink gadget can be repurposed to

```
ChainedExtractor cObj = new ChainedExtractor("execute");
LimitFilter lObj = new LimitFilter();
lObj.setBottomAnchor(cObj);
lObj.setComparator(cObj);

// Preparing object corresponding to the trigger gadget
BadAttributeValueExpException val = new BadAttributeValueExpException(lObj);

ObjectOutputStream os = new ObjectOutputStream(new
↪ FileOutputStream("payload.bin"));
// Writing the object into serialized bytestream (payload)
os.writeObject(val);
```

.

Listing 6: Simplified Java code showcasing the payload created by Crystallizer which uncovers the deserialization vulnerability in Coherence

mount an attack. Once the exploitation strategy for a sink is figured out, this information can then be reused in all the other concretized chains that are targeting the same sink. On average, it took an experienced Java developer with knowledge of deserialization attacks less than 5 minutes per chain to validate their exploitability once the conditions for exploitation were identified.

The hybrid analysis methodology adopted by Crystallizer can suffer from false negatives, *i.e.,* not creating payloads for certain vulnerable chains that exist in a target. These false negatives may creep in from two main sources. First, bounded search up to a user-configurable maximum length inherently misses longer gadget chains. However, this can be addressed by increasing the maximum path length and allocating more computation time. Second, the capability of Crystallizer to concretize a gadget chain depends on the concretization module capabilities in solving chain constraints. In some instances (as shown for `Vaadin`), these constraints may correspond to the setup of the environment. We plan to investigate the concretization of such chains as a part of future work.

Uncovering a deserialization vulnerability in an application requires not only the presence of a vulnerable gadget chain but also an entry point where the application is deserializing untrusted data. We acknowledge that in the context of discovering vulnerabilities in an application, Crystallizer is semi-automated. While it can find vulnerable gadget chains automatically, it still requires a user to identify an end-point in the application deserializing data where the payload can be delivered. However, in the context of libraries, Crystallizer can automatically create payloads that trigger the vulnerability. While exploiting this vulnerability, requires an application to be using that library, it does not change the fact that the vulnerability in the library still exists. This view is in line with prior responses from library developers [13].

7 RELATED WORK

Rasheed et al. [47] leverage partial instantiation of gadget chains by relying on heap abstraction, and using a fixed set of sinks. Similar hybrid approaches were proposed by Cao et al (ODDFUZZ [10] and GCMiner [11]) to identify deserialization vulnerabilities in Java applications. A key difference with these works is that they require a pre-defined set of sinks as compared to Crystallizer, which automatically identifies sinks. Specifically, certain chains like those exhibiting DoS (Listing 4) or using unconventional sinks for RCE (Listing 3) cannot be found by ODDFuzz and GCMiner. The corresponding sinks for these chains are not treated as security-sensitive based on their predefined list. Unfortunately, at the time of writing, ODDFUZZ is not open-sourced and we were unable to reproduce results from GCMiner.

Pacheco et al. propose automatic techniques to instantiate objects [45] which can benefit Crystallizer in its object instantiation. We plan to explore them as future work. Gauthier et al. [25] propose an active mitigation to recognize malicious chains through Markov-based modeling, while Crystallizer is a testing tool to find deserialization vulnerabilities. Cristalli et al. [15] discussed other dynamic mitigation policies. Regarding DoS, Dietrech et al. [19] manually create a payload that, upon deserialization, triggers large call trees recursively leading to resource exhaustion. In contrast, Crystallizer automatically discovers DoS-like gadget chains.

Deserialization attacks also impact other languages like PHP and .NET. Dahse et al. [17] employ a static analysis-based method to detect PHP object injection (POI) chains, and Park et al. [46] extend POI construction with an automatic exploit generation technique, both yielding impressive outcomes. However, these approaches are closely tied to PHP and, rely on predefined sinks. In contrast, Crystallizer identifies sinks automatically. Moreover, Java's static typing imposes more stringent constraints on gadget chain concretization compared to PHP's dynamic typing. Shcherbakov et al. [49] uncover .NET-based deserialization vulnerabilities. by leveraging known vulnerable chains. In contrast Crystallizer's focus is new gadget chains. ObjectMap [30] is designed to identify deserialization errors in PHP and Java applications. It identifies the entry points of an HTTP request, then probes different inputs until a deserialization error arises. ObjectMap, however, explores malicious input without modeling the input spaces as a gadget graph nor including the notion of source/sink gadgets.

8 CONCLUSION

Deserialization vulnerabilities are common in complex distributed applications. We introduce a hybrid approach to automatically discover such deserialization vulnerabilities, highlighting incomplete checks when objects are deserialized in target applications. Our method uses static analysis to identify candidate gadget chains and dynamic analysis to generate concrete payloads to exercise gadget chains showing proof of a deserialization vulnerability. Crystallizer outperforms existing state-of-the-art tools in uncovering Java-based deserialization vulnerabilities and is shown capable of mounting attacks on popular real-world applications.

ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their precise and detailed feedback and Chibin Zhang for his help with analyzing related work. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No. 850868), AFRL under FA8655-20-1-7048, SNSF under PCEGP2_186974, and a generous gift from Oracle Labs.

REFERENCES

[1] Apache. 2022. Apache Commons Collections Library. https://commons.apache.org/index.html.

[2] Apache. 2022. Denylist for Java-based deserialization. https://github.com/apache/kafka/blob/trunk/connect/runtime/src/main/java/org/apache/kafka/connect/util/SafeObjectInputStream.java.

[3] Apache. 2022. Java deserialization in Apache Pulsar. https://pulsar.apache.org/docs/v2.0.1-incubating/functions/api/#java-serde.

[4] Apache. 2022. Kafka—Distributed event streaming platform. https://github.com/apache/kafka.

[5] Apache. 2022. Lack of serialization filtering in Apache Pulsar. https://github.com/apache/pulsar/blob/master/pulsar-functions/api-java/src/main/java/org/apache/pulsar/functions/api/utils/JavaSerDe.java.

[6] Apache. 2022. Pulsar—Distributed pub-sub messaging platform. https://github.com/apache/pulsar.

[7] AWS. 2022. What is Kafka? https://aws.amazon.com/msk/what-is-kafka.

[8] Baldeung. 2022. transient keyword in Java. https://www.baeldung.com/java-transient-keyword.

[9] Alexander Belokrylov. 2022. Java—popular enterprise coding language. https://www.forbes.com/sites/forbestechcouncil/2022/04/06/why-and-how-java-continues-to-be-one-of-the-most-popular-enterprise-coding-languages.

[10] S. Cao, B. He, X. Sun, Y. Ouyang, C. Zhang, X. Wu, T. Su, L. Bo, B. Li, C. Ma, J. Li, and T. Wei. 2023. ODDFuzz: Discovering Java Deserialization Vulnerabilities via Structure-Aware Directed Greybox Fuzzing. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society.

[11] Sicong Cao, Xiaobing Sun, Xiaoxue Wu, Lili Bo, Bin Li, Rongxin Wu, Wei Liu, Biao He, Yu Ouyang, and Jiajia Li. 2023. Improving Java Deserialization Gadget Chain Mining via Overriding-Guided Object Generation. In *Proceedings of the 45th International Conference on Software Engineering (ICSE '23)*.

[12] CodeIntelligenceTesting. 2022. Jazzer — AutoFuzz mode. https://www.code-intelligence.com/blog/autofuzz.

[13] Apache Commons Collections. 2023. Apache Commons Collections security report. https://commons.apache.org/proper/commons-collections/security-reports.html.

[14] Confluent. 2022. Kafka connectors serialization. https://www.confluent.io/blog/kafka-connect-deep-dive-converters-serialization-explained/.

[15] Stefano Cristalli, Edoardo Vignati, Danilo Bruschi, and Andrea Lanzi. 2018. Trusted Execution Path for Protecting Java Applications Against Deserialization of Untrusted Data. In *Research in Attacks, Intrusions, and Defenses*, Michael Bailey, Thorsten Holz, Manolis Stamatogiannakis, and Sotiris Ioannidis (Eds.). Springer International Publishing, Cham, 445–464.

[16] CyNation. 2017. Equifax Data Breach. https://cynation.com/the-equifax-data-breach/.

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

[18] Jeffrey Dean, David Grove, and Craig Chambers. 1995. Optimization of object-oriented programs using static class hierarchy analysis. In *European Conference on Object-Oriented Programming*. Springer, 77–101.

[19] Jens Dietrich, Kamil Jezek, Shawn Rasheed, Amjed Tahir, and Alex Potanin. 2017. Evil pickles: DoS attacks based on object-graph engineering. In *31st European Conference on Object-Oriented Programming (ECOOP 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[20] Frohoff. 2018. Beanutils GT chain. https://github.com/frohoff/ysoserial/blob/master/src/main/java/ysoserial/payloads/CommonsBeanutils1.java.

[21] Frohoff. 2018. Groovy GT chain. https://github.com/frohoff/ysoserial/blob/master/src/main/java/ysoserial/payloads/Groovy1.java.

[22] Chris Frohoff. 2022. ysoerial : A collection of known gadget chains found in java-based software. https://github.com/frohoff/ysoserial.

[23] Andrew Gainer-Dewar. 2022. Djikstra-like path enumeration algorithm for directed graphs. https://jgrapht.org/javadoc/org.jgrapht.core/org/jgrapht/alg/shortestpath/AllDirectedPaths.html.

[24] Carlos Cardoso Galhardo, Peter Mell, Irena Bojanova, and Assane Gueye. 2020. Measurements of the most significant software security weaknesses. In *Annual Computer Security Applications Conference*. 154–164.

[25] François Gauthier and Sora Bae. 2022. Runtime Prevention of Deserialization Attacks. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER '22)*.

[26] Ian Haken. 2021. Gadget Inspector: Static discovery of gadget chains. https://github.com/JackOfMostTrades/gadgetinspector.

[27] Jang. 2021. AspectJWeaver GT chain. https://github.com/frohoff/ysoserial/blob/master/src/main/java/ysoserial/payloads/AspectJWeaver.java.

[28] JFrog. 2022. Log4Shell vulnerability mounted using java deserialization. https://jfrog.com/blog/log4shell-0-day-vulnerability-all-you-need-to-know/#appendix-b.

[29] Vini Kanvar and Uday P. Khedker. 2016. Heap Abstractions for Static Analysis. *ACM Comput. Surv.* (2016).

[30] Nikolaos Koutroumpouchos, Georgios Lavdanis, Eleni Veroni, Christoforos Ntantogian, and Christos Xenakis. 2019. ObjectMap: Detecting insecure object deserialization. In *Proceedings of the 23rd Pan-Hellenic Conference on Informatics*. 67–72.

[31] Kullrich. 2018. Vaadin GT chain. https://github.com/frohoff/ysoserial/blob/master/src/main/java/ysoserial/payloads/Vaadin1.java.

[32] Kaiser Mathias and Jasinner. 2019. Apache Commons Collections GT chain. https://github.com/frohoff/ysoserial/blob/master/src/main/java/ysoserial/payloads/CommonsCollections5.java.

[33] Kaiser Mathias and Jasinner. 2019. Apache Commons Collections GT chain. https://github.com/frohoff/ysoserial/blob/master/src/main/java/ysoserial/payloads/CommonsCollections2.java.

[34] Alvaro Munoz and Schneider. 2018. Beanshell GT chain. https://github.com/frohoff/ysoserial/blob/master/src/main/java/ysoserial/payloads/BeanShell1.java.

[35] Alvaro Munoz and Christian Schneider. 2016. Serial Killer: Silently Pwning Your Java Endpoints. https://paper.bobylive.com/Security/asd-f03-serial-killer-silently-pwning-your-java-endpoints.pdf.

[36] NVD. 2017. Apache Struts RCE vulnerability. https://nvd.nist.gov/vuln/detail/cve-2017-9805.

[37] NVD. 2023. CVE-2020-2555. https://nvd.nist.gov/vuln/detail/CVE-2020-2555.

[38] Oracle. 2021. Interface Serializable. https://docs.oracle.com/javase/7/docs/api/java/io/Serializable.html.

[39] Oracle. 2022. Java Deserialization using readObject. https://docs.oracle.com/javase/7/docs/api/java/io/ObjectInputStream.html#readObject().

[40] Oracle. 2022. Java Serialization using writeObject. https://docs.oracle.com/javase/7/docs/api/java/io/ObjectOutputStream.html#writeObject().

[41] Oracle. 2022. Object class in Java. https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html.

[42] Oracle. 2023. classpath in Java. https://docs.oracle.com/javase/tutorial/essential/environment/paths.html.

[43] Oracle. 2023. Coherence library. https://www.oracle.com/java/coherence/.

[44] Soot Oss. 2022. Soot. https://github.com/soot-oss/soot.

[45] Carlos Pacheco and Michael D Ernst. 2007. Randoop: feedback-directed random testing for Java. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*. 815–816.

[46] Sunnyeo Park, Daejun Kim, Suman Jana, and Sooel Son. 2022. {FUGIO}: Automatic Exploit Generation for {PHP} Object Injection Vulnerabilities. In *31st USENIX Security Symposium (USENIX Security 22)*. 197–214.

[47] Shawn Rasheed and Jens Dietrich. 2020. A Hybrid Analysis to Detect Java Serialisation Vulnerabilities. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering* (Virtual Event, Australia) *(ASE '20)*. Association for Computing Machinery, New York, NY, USA, 1209–1213. https://doi.org/10.1145/3324884.3418931

[48] Hovav Shacham. 2007. The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security* (Alexandria, Virginia, USA) *(CCS '07)*. ACM, New York, NY, USA, 552–561. https://doi.org/10.1145/1315245.1315313

[49] Mikhail Shcherbakov and Musard Balliu. 2021. Serialdetector: Principled and practical exploration of object injection vulnerabilities for the web. In *Network and Distributed Systems Security (NDSS) Symposium 202121-24 February 2021*.

[50] TIOBE. 2022. Popular programming languages for development. https://www.tiobe.com/tiobe-index/.

[51] Flavio Toffalini, Mariano Graziano, Mauro Conti, and Jianying Zhou. 2021. SnakeGX: a sneaky attack against SGX Enclaves. In *International Conference on Applied Cryptography and Network Security*. Springer, Cham, 333–362.

[52] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 2010. Soot: A Java bytecode optimization framework. In *CASCON First Decade High Impact Papers*. 214–224.

Received 2023-02-02; accepted 2023-07-27