SerdeSniffer: Enhancing Java Deserialization Vulnerability Detection with Function Summaries

Xinrong Liu[1,2], He Wang[1,2(✉)], Meng Xu[3], and Yuqing Zhang[1,2,4]

[1] Hangzhou Institute of Technology and School of Cyber Engineering, Xidian University, Xi'an, China
xinrongliu@stu.xidian.edu.cn, hewang@xidian.edu.cn

[2] National Computer Network Intrusion Protection Center (NCNIPC), University of Chinese Academy of Science, Beijing, China
zhangyq@nipc.org.cn

[3] University of Waterloo, Waterloo, Canada
meng.xu.cs@uwaterloo.ca

[4] School of Cyberspace Security, Hainan University, Haikou, China

Abstract. Java deserialization vulnerabilities arise when unexpected data triggers dangerous function calls during deserialization processes. Current deserialization vulnerability detection faces challenges such as path explosion caused by polymorphism [29] in Java, leading to incomplete analysis and inefficiency.

In this paper, we present SerdeSniffer, a new Java deserialization vulnerability detection tool to address these challenges. SerdeSniffer is the first tool that employs a bottom-up function summarization technique to mitigate path explosion effectively. Specifically, SerdeSniffer uses function summaries during interprocedural analysis to effectively prevent multiple calculations in taint analysis and utilizes fixpoint computation to analyze issues related to function recursion and cyclic calls.

To avoid omissions in summary information, we use the over-tainting method in taint analysis and treat uninitialized variables as taint sources. We also merge the summary information of called functions to facilitate polymorphic analysis. Furthermore, *SerdeSniffer* includes a vulnerability detection algorithm that starts from dangerous functions and uses summary information and propagation rules in a bottom-up approach, and employs a sanitizer to eliminate ineffective propagation paths.

In comparative experiments based on ysoserial [23], a tool for generating payloads that exploit Java object deserialization, *SerdeSniffer* identified nine more historical gadget chains than other open source tools. In testing the latest versions of components, *SerdeSniffer* discovered three new gadget chains within 600 s, the longest being 14 nodes, two of these new chains confirmed by CVEs in the PUBLISHED state.

Keywords: Static Analysis · Taint Analysis · Java Deserialization Vulnerabilities · Function Summaries

J. Garcia-Alfaro et al. (Eds.): ESORICS 2024, LNCS 14984, pp. 174–193, 2024.
https://doi.org/10.1007/978-3-031-70896-1_9

1 Introduction

Serialization refers to the process of converting the state of an object (typically represented as structured data) as storable or transmittable form. Serialization is commonly used for preserving runtime state, data storage, or remote procedure calls (RPC) scenarios [22]. Moremove, deserialization transforms serialized binary or string data back into a valid object. In Java programs, native serialization is accomplished through the `writeObject` method of a serializable class, and deserialization through the `readObject` method.

Native Java deserialization is not entirely secure due to polymorphism and class inheritance mechanisms in Java, which allow class field types to be replaced during the deserialization process. This can trigger unexpected subclass method calls. When applications handle serialized data from unreliable sources, attackers exploit these vulnerabilities by tampering with or supplying malicious serialized data as an attack vector, leading to remote code execution, data leaks, and service disruptions. In recent years, widely used components such as Fastjson [1] and the Java deserialization gadget chains in Apache Commons Collections [2] have highlighted this issue, with these gadget chains often being difficult to analyze manually. The attack surface typically covers all parts of the application that handle untrusted serialized data, particularly those dependent on external libraries. Unsafe components can compromise the security of other components or even the entire application, as seen in the command execution vulnerability in WebLogic that depends on multiple components [13].

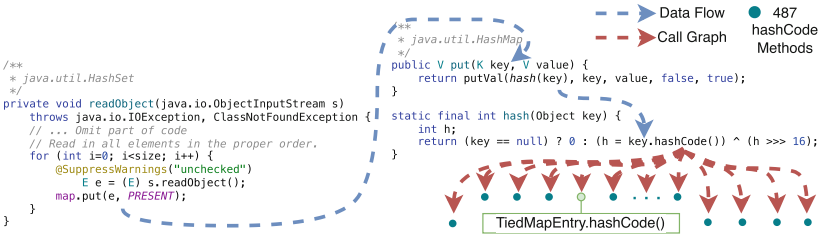

Fig. 1. Core code of the CC6 gadget chain in version 3.1 of the Apache Commons Collections library.

Figure 1 highlights a deserialization vulnerability in version 3.1 of the Apache Commons Collections library and demonstrates the path explosion due to Java polymorphism. Data flow shows the `key` variable in `key.hashCode`, generated during deserialization in `HashSet`, lacks static verification. As a result, any of the 487 subclasses with a method signature matching `hashCode` might be invoked, though typically only `hashCode` of `TiedMapEntry` triggers the CC6 gadget chain. The `this` variable in the `hashCode` function is also generated through the deserialization process, leading to path explosion in functions that are invoked starting from `this` variable. In addition to the polymorphism mentioned above, path

explosion is also exacerbated by recursive and cyclic function calls and further worsened by multi-component combination analysis.

Recent efforts in security research have integrated program analysis techniques to detect Java deserialization vulnerabilities, aiming to pinpoint potential gadget chains. Yet, balancing enhanced analytical precision with efficiency and handling path explosion remains challenging. For instance, GadgetInspector [25] utilizes the ASM framework and static taint analysis to construct function call graphs but limits the analysis length due to path explosion, leading to missed detections. Similarly, Serhybrid [33], despite starting from *MagicMethods* and employing pointer analysis, overlooks polymorphism issues and inadequately addresses path explosion. JDD [20] combines static taint analysis and pointer analysis to identify potential entry points and latent gadget fragments, yet still contends with the challenges of path explosion when addressing polymorphism.

1.1 Motivation

Despite recent progress in Java deserialization vulnerability detection solutions still have several key shortcomings. These limitations affect the efficiency and accuracy of vulnerability detection which eventually leads to false negatives. The motivation of this study is to address several core issues that are widely neglected or not fully resolved in current research and tool implementations:

(a) **Insufficiencies in Deserialization Analysis**

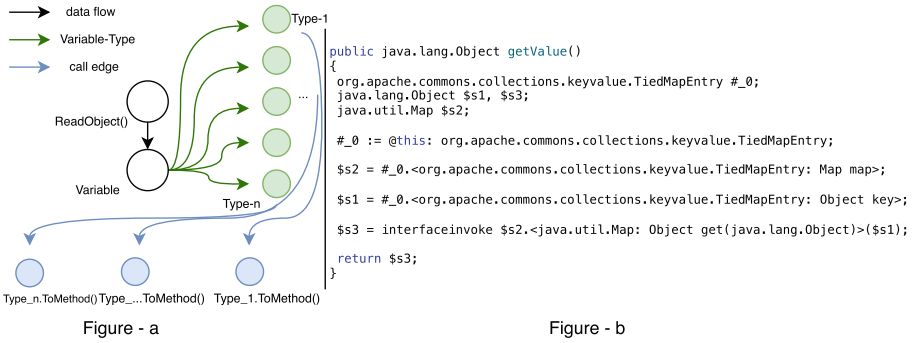

Fig. 2. Polymorphic issues in deserialization and disruption in data flow analysis.

Deserialization analysis must address Java polymorphism and the simulation of value assignments for uninitialized variables. During deserialization, objects are generated from data streams *without clear type information*, leading to multiple potential call paths and polymorphic issues, as illustrated in Fig. 2.a. The uncertainty in deserialized objects' contents and the absence of

object pointers in standard analysis disrupt pointer and taint analysis connections, as shown in Fig. 2.b. Conventional pointer analysis struggles without explicit `Store` and `New` assignments, impeding effective analysis between variables $s2$ and $s1$, and blocking data flow from `this` to $s3$.

(b) **Balancing Precision and Efficiency**
Achieving precision and efficiency in Java deserialization vulnerability detection is crucial. Polymorphic analysis and mock for uninitialized variables enhance accuracy and reduce efficiency. While taint analysis improves precision due to data flow tracking, it reduces efficiency. Traditional methods like the RTA algorithm [15], unsuitable for unknown deserialization object types, are replaced by the CHA (class hierarchy analysis) algorithm [22] to address polymorphism, enhancing analysis depth but expanding the space. To optimize efficiency without losing precision, CHA is applied selectively to tainted objects, conserving resources and time while ensuring high accuracy.

(c) **Challenges in Reflective Analysis**

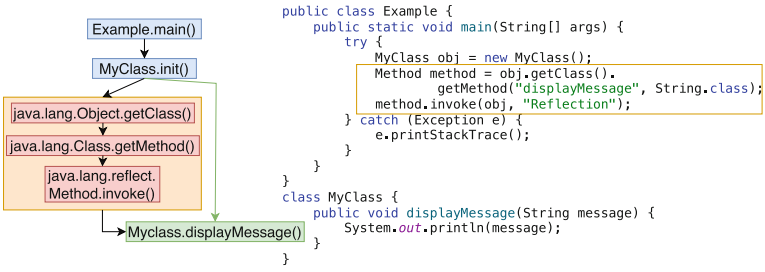

Fig. 3. Example of Java reflection in code.

The reflection complicates deserialization vulnerability detection by allowing runtime inspection, invocation, and modification of class properties and methods, as illustrated in Fig. 3. Programs can dynamically invoke methods like `displayMessage` without direct reference, enhancing development flexibility while posing significant challenges for static analysis. The unpredictable nature of reflection at compile time hinders accurate vulnerability detection, increasing the risk of false negatives, and complicating the comprehensive identification of security vulnerabilities in static analysis tools.

1.2 Research Contributions

This study introduces a new tool for detecting Java deserialization vulnerabilities, *SerdeSniffer*. This tool combines static taint analysis and graph databases to address the deficiencies in efficiency, accuracy, and depth of vulnerability mining due to the path explosion. The main contributions of this research include:

(a) **Use of Function Summaries:** *SerdeSniffer* introduces function summaries for the first time to address the path explosion introduced by polymorphism in Java deserialization vulnerability detection. *SerdeSniffer* proposes the BIFSum(Bottom-up Information Flow Summary) algorithm to systematically build function summaries within components. In the analysis process, BIFSum treats uninitialized variables as temporary taint sources and uses over-tainting analysis to prevent omissions in summary information. It also employs abstract fixpoint analysis to resolve function recursion and looping issues within the summarization process.
(b) **Rule-Based Vulnerability Detection:** *SerdeSniffer* employs rule-based vulnerability detection algorithms that analyze from dangerous functions using function summaries to determine if reachable functions can taint Sink functions. If taint rules are met, the reachable function controls the Sink. During analysis, data flow is sanitized to remove invalid variables. After retrieving the shortest gadget chain via a graph database, it verifies the type consistency of alias objects within the chain to minimize false positives.
(c) **Experimental Validation and Practical Application:** In comparative experiments with ysoserial, *SerdeSniffer* demonstrated high efficiency and effectiveness in detecting historical vulnerabilities, discovering 8 more historical gadget chains than other tools. In testing the latest versions of components, *SerdeSniffer* was able to identify three new gadget chains with a maximum length of 14 within 600 s, two of which have been confirmed in the CVE database. Experiments also show that *SerdeSniffer* can effectively detect vulnerabilities within a reasonable time frame even in scenarios involving complex components or component combinations.

1.3 Structure of the Paper

This paper introduces the *SerdeSniffer* framework and its advancements in detecting Java deserialization vulnerabilities. It starts with an overview of the background and the objectives. The second section describes the architecture and key technologies of *SerdeSniffer*, followed by a detailed explanation of its static analysis algorithms like the BIFSum algorithm in the third section. The fourth section evaluates *SerdeSniffer* through case studies, while the fifth analyzes the results, discussing its strengths and limitations. The sixth section reviews related work in the field of Java deserialization vulnerability detection, highlighting the advantages of *SerdeSniffer* relative to other technologies. Finally, the contributions of *SerdeSniffer* are summarized, and future research directions are discussed.

2 Framework Overview

SerdeSniffer is a static analysis framework designed for the detection of Java deserialization vulnerabilities. As shown in Fig. 4, the framework comprises two main parts: data processing and static analysis. The data processing part is

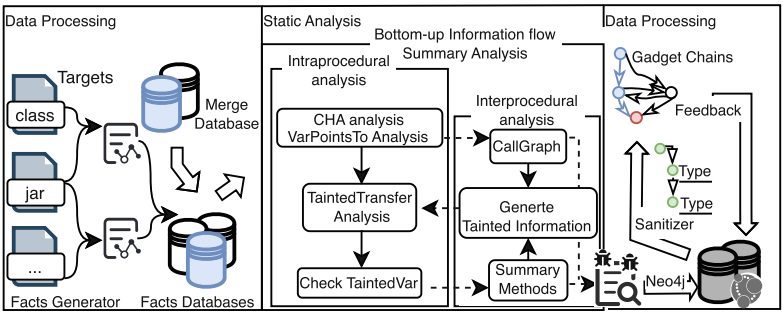

Fig. 4. The *SerdeSniffer* Framework Diagram

responsible for preparing, analyzing, and organizing data before and after static analysis; the static analysis part focuses on the BIFSum algorithm and vulnerability detection implemented based on Datalog [34], with the detailed algorithms being thoroughly introduced in Sect. 3.

Data Processing. The data processing component of *SerdeSniffer* is essential for handling data preparation, organization, and analysis. It converts Class, Jar, and other test targets into analyzable formats using the Soot framework [28], which generates SSA-formatted IR [26] that feeds into a Facts database for Datalog-based static analysis, as shown in Fig. 4. For combined components, *SerdeSniffer* merges their Facts databases to serve as new analysis inputs.

Post-analysis, data undergoes systematic organization in a graph database during the data organization phase. This is followed by a detailed analysis phase using predefined queries to generate function call graphs. During the data analysis phase, Sanitizers assess the type consistency of alias objects within the call graph. Detected inconsistencies prompt modifications to the query statements, refining them to exclude invalid call paths. This iterative process continues until the graph database yields no further results, ensuring that only valid and effective deserialization vulnerability gadget chains are recognized by *SerdeSniffer*.

Static Analysis. Static analysis is central to the *SerdeSniffer* framework, which includes intraprocedural analysis, interprocedural analysis, and vulnerability detection, all implemented via Datalog-based fixpoint analysis. Below is an overview of functionalities in each component.

During intraprocedural analysis, static analysis initially performs pointer analysis and taint analysis on the SSA-formatted Intermediate Representation (IR). This includes analyzing the preliminary call graph and intraprocedural data flow information for each function. The system then checks if the `this` pointer and return values of the function might be tainted by input values to build function summaries. For tainted function call instructions, the CHA algorithm is used to build function call graphs and address polymorphism issues.

Interprocedural analysis relies on function summaries and call graphs to perform cross-functional taint analysis. In this phase, static analysis calculates new tainted variables based on the summaries of called functions and updates taint propagation information. Through fixpoint analysis, the system continually identifies and updates new tainted variables, iterating repeatedly until all relevant function summaries are fully analyzed to ensure comprehensive analysis.

The vulnerability detection phase is based on the results of the first two analysis stages. By analyzing the constructed call graphs and taint propagation information, the system identifies potential vulnerability gadget chains. Using predefined rules for dangerous function propagation, the reachable and propagable dangerous function call graph is constructed from the bottom up. Once the analysis is complete, this information is imported into a graph database to support the final identification and assessment of vulnerabilities.

3 Algorithms

This chapter introduces the Bottom-up Information Flow Summary (BIFSum) Algorithm used in *SerdeSniffer*, which addresses the path explosion. It also details the data processing part of *SerdeSniffer*, explaining the combination analysis between components and data cleansing to enhance accuracy.

3.1 Bottom-Up Information Flow Summary (BIFSum)

The BIFSum addresses several challenges in deserialization analysis, including the balance between precision and efficiency, and the difficulties of reflective analysis. This method specifically targets the unique nature of object initialization during deserialization, where initialization is conducted via deserialization rather than through constructors. Traditional methods often fail to adequately address the complexity introduced by these uniquely initialized objects, and the resulting path explosion significantly limits the comprehensive and efficient analysis of the code space.

BISum mitigates path explosion by establishing function summaries. During intraprocedural analysis, function parameters and the **this** are designated as taint sources. Depending on whether the Receiver Object is tainted, CHA algorithm or pointer analysis is executed to construct a call graph. Analyzing the possibility that taint sources may reach the variables of the called functions, thus generating **TaintedToInvocationVariable** as function summary information in Algorithm 1. In interprocedural analysis, the pre-computed summaries are used to directly uupdate the taint analysis information in the caller, facilitating the calculation of new taint propagation objects until all functions are analyzed and no new summary information emerges.

Algorithms Description. The main content of the BIFSum algorithm is detailed in Algorithm 1. Initially, all functions that require summary computation are identified, labeling methods with no function calls as `BottomMethod`

and others as `NodeMethod`. Lines 5–7 in Algorithm 1 are dedicated to the initialization phase for each function, where the `this` pointer and formal parameters of the current function are treated as sources of taint. Fields and arrays that cannot be initialized are simulated and set as temporary taint sources to prevent disruptions in pointer analysis and taint analysis.

Algorithm 1. BIFSum (Bottom-up Information Flow Summary)

```
1: def Methods: Collection of all Methods
2: def SourceVariable: This and FormalParams
3: def LostVar: Variables that lose value in some methods
4: while !fixPoint.reachable() do
5:     CurrentMethod ←Methods.pop()
6:     TSrc.add(CurrentMethod.SourceVariable)
7:     TmpTSrc.add(CurrentMethod.LostVar)
8:     TVar ←P/Taint.Analysis()
9:     if TmpTSrc.isTransferredFrom(TSrc) then
10:        Merge(TmpTSrc, TSrc)
11:    end if
12:    for CurrentStmt: CurrentMethod.stmts do
13:        if CurrentStmt.instanceof(VirtualInvocation)
             and TVar.contains(CurrentStmt.base) then
14:            ToMethod ←CHA(baseType,MethodSignature)
15:        else
16:            ToMethod ←VarPointsTo_CallGraph(baseVar,MethodSignature)
17:        end if
18:        if CurrentStmt.isReflectable(VirtualInvocation) then
19:            ReflectMethod, ReflectClass ←VarPointsTo(PreviousVariables)
20:            ToMethod ←Extend_Reflect(ReflectMethod, ReflectClass)
21:        end if
22:        Update(TVar, ToMethod.summary)
23:        CallGraphInSummarizer ← CallGraph(CurrentMethod, ToMethod)
24:    end for
25:    if TVar.transferred(TSrc) and TVar.isRetVars() then
26:        Summary(TaintSouTSrcrce, TVar)
27:    end if
28: end while
```

Datalog Rules abot Function Summary Information:

```
TaintedToInvocationVariable(from, from_method, to, to_method) :-
  (TaintedToInvocationParam(from, from_method, to, to_method);
  TaintedToInvocationBase(from, from_method, to, to_method))
```

In actual pointer and taint analyses, line 8 specifies using the P/Taint Analysis algorithm based on over-tainting to perform intraprocedural analysis and avoid summary omissions. Unlike traditional analyses, P/Taint analysis independently of pointer analysis [24], enhancing speed. After initial analyses, lines 9–11 assess whether taint sources can propagate temporary taint sources; if so, taint

information is merged. Over-tainting in P/Taint Analysis extends tainted elements to entire arrays or objects, ensuring completeness in function summaries.

BIFSum algorithm has two approaches for interprocedural call analysis: when the base of a call instruction is a tainted object, the CHA algorithm is used to avoid underreporting caused by deserialization assignments, as in lines 12–14; otherwise, pointer analysis is utilized to compute call information, as in lines 15–17. After obtaining call information, as in lines 22–23 of the algorithm, the summary information from the `ToMethod` is used to compute variable propagation information. Lines 25–27 focus on computing function summaries, particularly detecting which taint sources affect return variables like `this` and `return`.

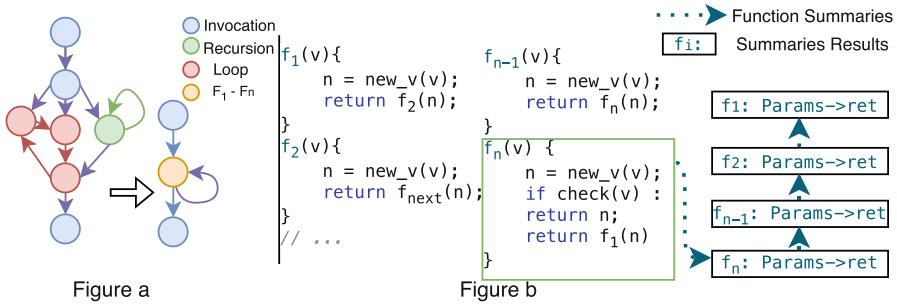

Fig. 5. a: Abstract representation of cycles and recursion; b: Abstract function taint propagation and function summary.

Cycles and Recursion. In static analysis, when dealing with issues of recursion and loops, the BIFSum algorithm employs an abstract fixpoint analysis method. As shown in Figs. 5-a and 5-b, BIFSum simplifies recursive and cyclic calls into a series of function calls from F_1 to F_n in the function call graph, where n represents the depth of recursion or number of cycles, and it is assumed that these calls include a termination condition in F_n.

When F_n meets the termination condition and its return value ret is potentially influenced by its taint sources (such as $this$ and $formalParams$), we will process a function summary for F_n, as shown in Fig. 5-b, depicting $F_n : Param \rightarrow Ret$, indicating that function parameters can taint the function return value. During the function summary calculation process illustrated in Fig. 5-b, BIFSum applies the fourth line of the algorithm for fixpoint analysis. The generated $Summary-F_n$ is used to infer upward from F_1 to F_{n-1}, continuing until all functions are analyzed. This method optimizes the analysis process and avoids performance issues common in traditional recursive or cyclic analyses, such as repeated calculations in path explosion.

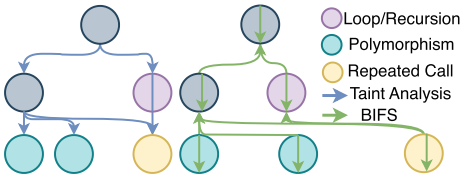

Fig. 6. Comparison of the BIFSum Algorithm and Taint Analysis

Handling Path Explosion. Figure 6 illustrates specifically how the BIFSum algorithm in *SerdeSniffer*, compared to other static analyses, utilizes function summaries to address the path explosion. Initially, the BIFSum algorithm employs function summaries in a bottom-up manner only during interprocedural analysis; for intraprocedural analysis, it computes and stores function summaries independently of other functions and only queries them when using the summaries. Additionally, function summaries include mappings from taint sources to taint outputs, which reduces the number of states that need to be considered, abstracts away the internal details of functions, and focuses on how taint is propagated through functions.

As depicted in Fig. 6, functions that are repeatedly called in taint analysis undergo complete pointer and taint analysis for each of the n different calling edges, leading to n full analyses for the repeatedly called function parts. In contrast, for the function summaries of BIFSum, once summaries are constructed based on over-tainting for the repeatedly called functions, the analysis of n calls merely utilizes the mapping from taint sources to taint outputs, without redoing the pointer and taint analyses.

Moreover, concerning function calls induced by Java polymorphism in taint analysis, it is necessary first to compute the calling function, and then perform full taint analysis among the m polymorphic target functions. For the BIFSum algorithm, it simply queries the summary information of the m target functions, eliminating the need to redo taint analysis.

Reflective Analysis. BIFSum employs a straightforward reflective analysis to address reflection issues in programs. In the program, BIFSum uses pointer analysis to determine if the parameter in `forName` points to a string constant, obtaining the class name of the target function, as in lines 18–21 of the Algorithm 1. Then, by assessing if the parameter in `getMethod` points to a string constant, it obtains the function name of the target function. Finally, based on the class name and function name, it locates the function and uses line 23 of BIFSum to invoke the summary information of the reflective function, updating variable propagation information within the function.

Utilizing Summaries for Vulnerability Detection. This section details how the BIFSum algorithm utilizes the TaintedToInvocationVariable information for vulnerability detection. The vulnerability detection process involves two

key steps: establishing rules for `Sink` functions and conducting a bottom-up vulnerability detection based on these rules. For instance, lines 2–4 in Algorithm 2 define a `SinkSummarizerRule` for command execution, stating that the controllability condition for the `Sink` function is that the first formal parameter is propagative, and this rule is designated as Rule 1. Moreover, BIFSum establishes rules for common high-risk functions, including controllable reflection functions, file reading, command execution, class loading, RMI, and JNDI.

Algorithm 2 . Datalog rules for Sink Summarizer and Method Summarizer

```
1: Datalog Rules:
2: SinkSummarizerRule(sink, 1) :-
3:    ReachabelSinkingMethod(source, _, sink, one), FormalParam(0, sink, one),
4:    sink = "<java.lang.Runtime: java.lang.Process exec(java.lang.String)>".
5: ReachableMethod(Method, from, Sink, to):-
6:    ((TaintedToInvocationVariable(from, Method, to, Sink), DefineSink(Sink));
7:    (ReachableMethod(PMethod, PFrom, SinkMethod, to),
   TaintedToInvocationVariable(from, Method, PFrom, PMethod))).
8: Result(method, sink, rule) :-
9:    ReachableMethod(method, from, sink, one), rule = 1, isSourceFrom(from),
10:   FormalParam(0, sink, one), SinkSummarizerRule(sink, rule).
```

After establishing the rules for the `Sink` functions, as shown in lines 5–7 of Algorithm 2, BIFSum begins from the `Sink` function and calculates the propagability of function parameters to the `Sink` function from the bottom up, until all function parameters to the `Sink` function are confirmed to be reachable and controllable, and this step is referred to as `ReachableMethod`. Then, lines 8–10 specify that under Rule 1, if the `from` variable in `ReachableMethod` is a predetermined source of taint, such as the `this` variable in the `readObject` function, and fully satisfies the conditions set by the `SinkSummarizerRule`, BIFSum considers the current `ReachableMethod` able to effectively trigger and control the `Sink` function. Lastly, all reachable and controllable call graphs from `ReachableMethod` to the `Sink` function are imported into the Neo4j database.

3.2 Data Processing

Combination Analysis. In the component combination analysis, the *Serde Sniffer* framework defines the Facts database for each component as $Facts_n$, where each database contains multiple relations, denoted as $Facts_R ules_m$. The variable m represents the number of relation types that need to be processed, including information on variables, details of assignments, function calls, and other key relational information. During the combination process, the actual operation involves merging the corresponding $Facts\_Rules_m$ from each $Facts_n$ to create a new consolidated database, $Facts\_Merge$. This process can be described as:

$$Facts\_Merge = \bigcup_{i=1}^{n} \bigcup_{j=1}^{m} Facts\_Rules_{ij} \tag{1}$$

In this merging process, each entry of $Facts\_Rules_m$ includes the component package information, and all variables are treated in the Static Single Assignment (SSA) form. This approach ensures that data do not overwrite each other during the merge, thereby maintaining data integrity and consistency.

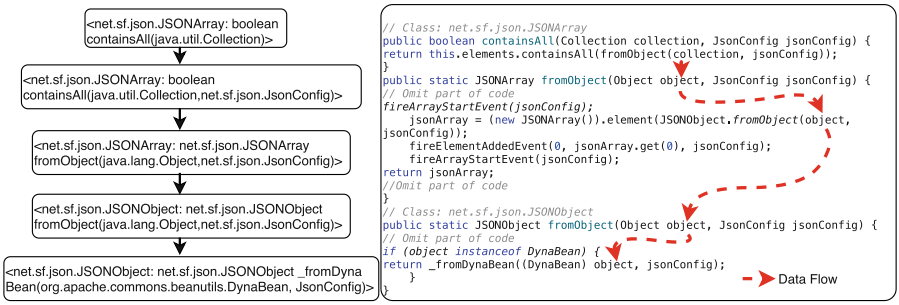

Fig. 7. Example of Type Inconsistency Detection

Data Cleansing. After the static analysis is complete, *SerdeSniffer* imports all effective call graphs that can trigger vulnerabilities into a graph database. Within this database, deserialization functions marked as the start method (StartMethod) are used to find the shortest path graphs from StartMethod to dangerous functions, which are subsequently output as results.

Upon obtaining specific gadget chain call graphs, *SerdeSniffer* utilizes the call graph information to detect the consistency and validity of deserialized objects based on the data flow information, from top to bottom. Specifically, the system analyzes data flows based on interprocedural and intraprocedural taint analysis results, and checks the inclusiveness of different object types under the same pointer from top to bottom. If the type of a previous object is a superclass or the same class of a subsequent object, it is determined that the types of these objects are inclusive, thereby confirming their consistency.

For example, in the results of testing version 2.14.0 of `org.zaproxy.zap`, Fig. 7 shows the core code and key call graph of a false positive gadget chain. In this graph, the collection object in the `containsAll` method and the object in `fromObject` alias. The type of the collection is the `Collection` class, while the actual type of the object is the `DynaBean` class. Since there is no class inheritance relationship between these two classes, there is a type conflict between the two variables, leading to the determination that this call chain is invalid.

4 Experiments

To effectively implement and evaluate the algorithms in *SerdeSniffer*, we integrated the BIFSum algorithm within the premier Java static analysis tool, Doop [17]. Doop leverages the Datalog language, utilizing frameworks like soufflé [27], to perform pointer and taint analysis on large-scale Java programs [16]. Rules files in Doop, which are based on Soufflé, are extendable, allowing for the implementation of the BIFSum algorithm by adding analysis rules.

During the implementation of the BIFSum algorithm in Doop, we first modified the `Soot-fact-generator` in Doop to accommodate as many JDK versions as possible, testing a broader range of components. Additionally, in the analysis rules of Doop, we disabled the original pointer and taint analyses that were extended from the call graph, such as ContextResponse and StaticContextResponse. Instead, we utilized `CallGraphInSummarizer` from *SerdeSniffer* to ensure intraprocedural pointer and taint analyses and interprocedural function summary utilization. Moreover, *SerdeSniffer* expands the analysis scope to all functions and provides lightweight function summary information for native functions to ensure as many functions as possible generate summary information.

In the data processing stage, *SerdeSniffer* uses Neo4j for graph database implementation and shortest path retrieval. As one of the most advanced graph database engines, Neo4j allows efficient data import through the database import method, minimizing the impact of data import processes [8]. *SerdeSniffer* applies the Dijkstra algorithm to efficiently identify the shortest paths in deserialization vulnerability gadget chains [30]. In subsequent data processing, *SerdeSniffer* designs customizable blacklist query statements to retrieve the next available gadget chain when issues arise with deserialized objects or functions.

4.1 Experimental Setup

The experiments were conducted on an Intel(R) Xeon(R) Gold 5117 CPU @ 2×2.0GHz with 90G of memory, running on Debian 12.2. The experimental analysis utilized Doop version 4.24.10, deployed in an analysis environment with soufflé version 2.3 and Oracle JDK 8. The choice of Oracle JDK 8, despite being an older version, was due to its widespread adoption and compatibility with many legacy systems, ensuring comprehensive and relevant test coverage. Analysis options in Doop include context-insensitive analysis, minimized information flow analysis, lightweight reflection, and simple proxy options.

4.2 Test Dataset

To evaluate the effectiveness of SerdeSniffer, we chose ysoserial as the primary baseline dataset. The ysoserial project is a Java deserialization vulnerability exploit project and is the most popular and prominent Java deserialization exploit tool. It includes major and exploitable Java deserialization exploit chains, which have been widely used and studied, becoming an important benchmark in the field of Java deserialization vulnerability detection. The project contains

31 historical vulnerability components, 9 of which rely solely on their own components, while the remaining 22 require other components within ysoserial for exploitation, as detailed in Table 1.

4.3 Effectiveness

The experimental results of SerdeSniffer on the test dataset are shown in Table 1. This table shows the number of historical gadget chains pre component in the `Count` column, and the `Results` column compares the number of effective exploitation chains identified by SerdeSniffer to the total number of gadget chains analyzed. Additionally, for larger components like clojure, *SerdeSniffer* not only completes the analysis but also effectively identifies historical gadget chains. Even when analyzing deserialization vulnerabilities involving multiple components, *SerdeSniffer* maintains effective performance within acceptable time frames.

Table 1. *SerdeSniffer* Experimental Results. Note: "R" for Reachable and "T" for Taintable

Component	Version	Time(s)	Loc	Objects	R	T	Count	Results
ysoserial Single Component								
commons-collections	3.1	325	5.53M	52	33	11	5	5/11
commons-collections4	4.0	211	3.12M	56	39	13	2	2/13
clojure	1.8.0	450	6.71M	18	25	10	2	2/10
rome	1.0	307	5.49M	18	10	5	1	1/5
bsh	2.0b5	285	5.50M	3	19	1	1	0/1
groovy	2.3.9	650	8.09M	19	6	0	1	0/1
c3p0	0.9.5.2	306	5.93M	32	15	9	1	1/9
rhino.js	1.7R2	305	5.76M	27	5	1	1	1/1
jython-standalone	2.5.2	856	9.31M	44	105	36	1	1/36
ysoserial Component Combination								
spring-beans; core; xalan	4.1.4.R; 2.7.2	1458	7.27M	29	23	1	1	1/2
json-lib; logging, lang; aop, core; beanutils;	2.4; 1.0; 1.2; 4.1.4.R; 1.9.2	1365	6.45M	61	33	11	1	1/11
Discovered vulnerabilities								
clojure	1.12-alpha5	579	6.88M	20	30	15	2	2/15
beanutils;json-lib;cc;aop	1.9.4; 2.4; 5.3.32; 3.2.2	1242	11.22M	27	15	5	1	1/5

Compared to previous tools, *SerdeSniffer* can identify more historical gadget chains. Table 2 compares the performance of tools such as SerHybrid, GadgetInspector and ODDFuzz [9] on different components. The `Count` and `Results` columns in Table 2 correspond to those in Table 1, indicating historical vulnerability counts and comparing effective chains identified by different tools. In testing the `commons-collections 3.1` component, *SerdeSniffer* identified 5 effective gadget chains in 325 s, analyzing a total of 11 potential chains; in contrast, *SerHybrid* and *GadgetInspector* respectively identified only 1 effective gadget chain. Additionally, *ODDFuzz* identified only 3 effective gadget chains. Additionally, within the effective time frame, *SerdeSniffer* can effectively identify historical exploitation chains in components such as `bsh 2.0b5`.

Table 2. Comparative Experiments. Note: "TO" stands for "TIMEOUT", "T" for Time in seconds and "R" for Results

Component	Version	Count	*SerdeSniffer*		SerHybrid		GadgetInspector		ODDFuzz	
			T	R	T	R	T	R	T	R
commons-collections	3.1	5	325	5/11	1894	1/14	57	1/4	N/A	3/87
commons-collections4	4.0	2	211	2/13	837	1/15	55	0/4	N/A	2/112
clojure	1.8.0	2	450	2/10	TO	N/A	62	1/12	N/A	1/184
bsh	2.0b5	1	285	1/1	851	1/1	58	0/2	N/A	0/8
jython	2.5.2	1	856	1/36	TO	N/A	77	0/42	N/A	1/32
rome	1.0	1	307	1/5	578	0/1	56	0/2	N/A	1/5

4.4 Vulnerability Discovery

SerdeSniffer unveiled fresh insights during tests on recent components. Specifically, in Clojure version 1.12-alpha5, it detected a command execution gadget chain after 579 s of analysis, assigning it CVE-2024-22871 [4] and CVE-2017-20189 [3], with an introduction to the latter provided in Appendix A.

And, it is important to note that CVE-2017-20189 applied for in 2024, was found to affect Clojure version 1.9 and had a similar historical gadget chain in 2017 [11]. Therefore, the final assignment of CVE-2017-20189 includes both the newly discovered gadget chain by SerdeSniffer and the historical gadget chain [3]. Additionally, it was observed that the dependent components of zaproxy could combine to form a new deserialization gadget chain with spring-aop [5].

5 Discussion

According to the experimental results presented in Table 1, *SerdeSniffer* has demonstrated its effectiveness across multiple components, successfully identifying known vulnerabilities in components including Clojure and commons-collections. The *SerdeSniffer* tool has shown its capability in the static analysis domain for effective detection of Java deserialization vulnerabilities, particularly the BIFSum algorithm, which effectively addresses the path explosion. The Sanitizer mechanism also helps to avoid variables and statements that cannot propagate, thereby enhancing the precision of the analysis.

According to the comparative experimental results in Table 2, *SerdeSniffer* was compared against some tools. The results indicate that *SerdeSniffer* performs well, particularly in handling complex components and large-scale Java programs, confirming its effectiveness and practicality in detecting Java deserialization vulnerabilities.

Tests on groovy and bsh components revealed that historical vulnerabilities, involving dynamic proxies used in dangerous functions, couldn't be detected. As shown in Fig. 8, the BeanShell1 gadget chain exploited dynamic Comparator

proxies, binding malicious `InvocationHandler` implementations to the comparator of PriorityQueue. The framework of *SerdeSniffer* struggles with such proxies, leading to inaccuracies in vulnerability detection. Future work could explore an analysis of class inheriting `InvocationHandler` and `Serializable` for better reachability and propagability, and utilize runtime data to refine static analysis of dynamic proxies and reflection.

```
public PriorityQueue getObject(String command) throws Exception {
    // BeanShell payload
    String payload =
     "compare(Object foo, Object bar) {new java.lang.ProcessBuilder(new String[]{" +
       Strings.join( Arrays.asList(command.replaceAll("\\\\","\\\\\\\\")
       .replaceAll("\"","\\\"").split(" ")), ",", "\"", "\"")+"}).start();return new Integer(1);}";
    // Create Interpreter
    Interpreter i = new Interpreter();
    i.eval(payload);
    // Create InvocationHandler
    XThis xt = new XThis(i.getNameSpace(), i);
    InvocationHandler handler = (InvocationHandler) Reflections.
     getField(xt.getClass(), "invocationHandler").get(xt);
    // Create Comparator Proxy
    Comparator comparator = (Comparator) Proxy.
      newProxyInstance(Comparator.class.getClassLoader(),new Class<?>[]{Comparator.class},handler);
    // Prepare Trigger Gadget (will call Comparator.compare() during deserialization)
    final PriorityQueue<Object> priorityQueue = new PriorityQueue<Object>(2, comparator);
    Object[] queue = new Object[] {1,1};
    Reflections.setFieldValue(priorityQueue, "queue", queue);
    Reflections.setFieldValue(priorityQueue, "size", 2);
    return priorityQueue;
}
```

Fig. 8. BeanShell1 Historical Vulnerability Validation Code

Unlike mixed analysis tools such as SerHybrid and GCMiner [18], *SerdeSniffer* does not combine static and dynamic analyses, which restricts automatic vulnerability validation. Future enhancements could include constructing an abstract object graph from data flow and utilizing call graphs to guide the fuzzing process in tools like JQF [32] for generating test cases.

Despite the extensive research in the field of Java deserialization vulnerability detection, there are not many open-source tools available for comparison, as Table 3 shows the actual open-source status of recent works. In Table 3, GCMiner, although open-source, cannot be executed properly due to missing components such as fake-tabby, and there are relevant discussions in the GitHub community [7]. In addressing path explosion using Function Summaries, only summarizes dynamic functions, not considering the potential path explosion caused by the introduction of the CHA algorithm in normal method calls.

Table 3. Status of Recent Tools as of April 2024. Note: ● for True, ○ for False, and ⊙ for Special

Tool	Open Source	Polymorphic Analysis	Function Summaries	Dynamic Analysis	Data Generation
tabby [14]	●	●	○	○	○
GadgetInspector [25]	●	○	○	○	○
Serhybrid [12]	●	○	○	●	●
GCMiner [6]	⊙	○	○	●	○
ODDFUZZ [9]	○	●	○	●	●
JDD [20]	○	●	⊙	●	●
SerdeSniffer [10]	●	●	●	○	○

6 Related Work

To identify Java deserialization vulnerabilities, researchers initially used static analysis on Java apps. Ian Haken created GadgetInspector, the first open-source tool specifically for Java deserialization exploit chains. It leverages the ASM bytecode framework for taint analysis on binaries, allowing quick vulnerability detection within a practical time frame. GadgetInspector is widely recognized in both academia and industry and remains the top choice among security experts. However, it fails to address path explosion due to polymorphism and setting a fixed analysis duration limits the thoroughness of the analysis.

Subsequently, with the advancement of static analysis technology, researchers adopted more comprehensive and unified static analysis frameworks for Java deserialization vulnerability detection. In the Tabby Project [21], Java code is converted into a code property graph stored in a graph database. By analyzing data flows on top of detecting function call graphs, it checks for the reachability of dangerous functions. As the number of function call graphs increases, so does the analysis space and overhead of data flows, still facing the problem of path explosion.

Shawn Rasheed introduced the SerHybrid tool, which is based on the cutting-edge Java analysis tool Doop and was the first to use a hybrid analysis approach to inspect Java deserialization vulnerabilities. SerHybrid begins analyzing serialization issues from common functions such as `toString` and `hash`, but it overlooks the special nature of the serialization process and the polymorphism issues involved, leading to insufficient coverage of vulnerability detection. Additionally, SerHybrid employs Randoop for test data generation, but the original Randoop [31], as a unit testing tool, lacks the capability to generate complex objects.

Later, more researchers applied hybrid analysis to Java deserialization vulnerability detection. Sicong Cao and others introduced tools like ODDFuzz [19] and GCMiner during the same period. Unlike the hybrid methods in SerHybrid, ODDFuzz and GCMiner detail hybrid analysis and focus on the generation capability of verification data. Other researchers proposed the JDD tool, aimed at reducing the impact of path explosion by constructing gadget fragments from the bottom up, and addressing dynamic features such as reflection and proxies.

Compared to existing methods, *SerdeSniffer* introduces a function summary-based static analysis approach that effectively mitigates path explosion for the first time, balancing efficiency with accuracy. Additionally, *SerdeSniffer* provides a new static analysis tool for existing hybrid analysis researches, enriching current hybrid analysis methods. However, *SerdeSniffer* uses function call graphs and taint analysis, leading to a higher rate of false positives due to possible over-tainting and incomplete reachability of function call graphs. Future integration of hybrid analysis could lower false positives and improve automatic exploit generation efficiency. Additionally, *SerdeSniffer* struggles with special invocations like dynamic proxies, increasing the risk of false negatives.

7 Conclusion

This paper introduces *SerdeSniffer*, a novel framework for detecting Java deserialization vulnerabilities. Employing the innovative BIFSum algorithm, *SerdeSniffer* effectively resolves the path explosion caused by polymorphism in Java deserialization for the first time. By integrating a rule-based vulnerability detection algorithm and a Sanitizer mechanism, *SerdeSniffer* enhances the precision and efficiency of vulnerability assessments. This work offers a groundbreaking approach to mitigating Java deserialization vulnerabilities, providing a new tool for researchers to identify and address security gaps in Java applications.

Acknowledgement. This work was supported by National Key Research and Development Program of China (No.2023YFB3106400, 2023QY1202), the National Natural Science Foundation of China (U2336203, U1836210), the Key Research and Development Science and Technology of Hainan Province (GHYF2022010), and Beijing Natural Science Foundation (4242031).

A Appendix

The content of this appendix is the Clojure command execution gadget chain.

A.1 Clojure Command Execution

The new deserialization vulnerability in Clojure, affecting versions 1.9 to 1.12-alpha5, presents distinct characteristics compared to historical vulnerabilities. To address vulnerabilities, version 1.9 introduced patches that blocked the serialization of specific classes including `AbstractTableModel$ff19274a`. Despite these efforts, this version also inadvertently made `clojure.lang.Var` serializable, thereby opening up new avenues for potential exploitation.

SerdeSniffer discovered a new exploitation chain that bypasses these patch restrictions. As shown in Fig. 9, the deserialization of `HashMap` triggers the `hashCode` function in the `PersistentQueue` class. This sequence continues into

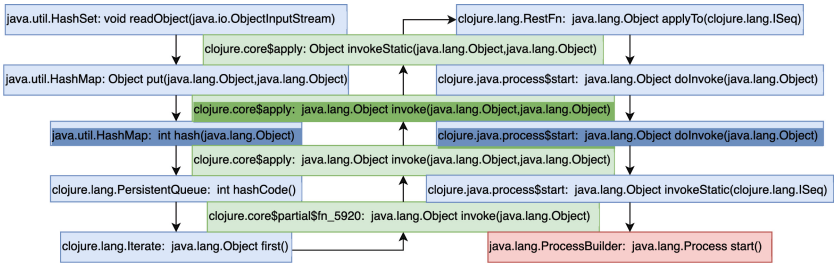

Fig. 9. Clojure Deserialization Vulnerability, Call Chain Length of 14

the `first` function of `PersistentQueue$Seq`, which allows for the serialization and manipulation of its `Iterate` objects. These serialized objects facilitate function calls that use the serialized `f` and `prevSeed` fields, ultimately leading to command execution via the `invoke` function of `core$partial$fn__5920` in `process$start`.

References

1. alibaba/fastjson. https://github.com/alibaba/fastjson
2. Collections - home. https://commons.apache.org/proper/commons-collections/
3. CVE-2017-20189. https://nvd.nist.gov/vuln/detail/CVE-2017-20189
4. CVE-2024-22871. https://nvd.nist.gov/vuln/detail/CVE-2024-22871
5. Gadget chains in zaproxy. https://bugcrowd.com/submissions/33b61ea8-30cd-40b2-81c8-30b5dff979d0
6. GCMiner/GCMiner: Artifact for ICSE 2023. https://github.com/GCMiner/GCMiner
7. Issues · GCMiner/GCMiner. https://github.com/GCMiner/GCMiner
8. neo4j/neo4j: Graphs for everyone. https://github.com/neo4j/neo4j
9. ODDFuzz/ODDFuzz. https://github.com/ODDFuzz/ODDFuzz
10. SerdeSniffer/SerdeSniffer. https://github.com/SerdeSniffer/SerdeSniffer
11. Snyk vulnerability database | snyk. https://security.snyk.io/
12. unshorn/serhybridpub. https://bitbucket.org/unshorn/serhybridpub/src/master/
13. Weblogic server|oracle. https://www.oracle.com/java/weblogic/
14. wh1t3p1g/tabby: Code analysis tool. https://github.com/wh1t3p1g/tabby
15. Allard, A., Albinsson, B., Wadell, G.: Rapid typing of human adenoviruses by a general PCR combined with restriction endonuclease analysis. J. Clin. Microbiol. **39**(2), 498–505 (2001)
16. Antoniadis, T., et al.: Porting doop to soufflé: a tale of inter-engine portability for datalog-based analyses. In: Proceedings of the 6th ACM SIGPLAN Int. Workshop on SOAP, pp. 25–30. ACM (2017). https://doi.org/10.1145/3088515.3088522
17. Bravenboer, M., et al.: Strictly declarative specification of sophisticated points-to analyses. In: OOPSLA09, pp. 243–262. ACM (2009)
18. Cao, S., et al.: Improving java deserialization gadget chain mining via overriding-guided object generation. In: 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), pp. 397–409. IEEE (2023)

19. Cao, S., et al.: ODDFuzz: Discovering java deserialization vulnerabilities via structure-aware directed greybox fuzzing. In: 2023 IEEE Symposium on Security and Privacy (SP), pp. 2726–2743. IEEE (2023)
20. Chen, B., et al.: Efficient detection of java deserialization gadget chains via bottom-up gadget search and dataflow-aided payload construction. In: 2024 IEEE Symposium on Security and Privacy (SP), pp. 150–150. IEEE Computer Society (2024)
21. Chen, X., et al.: Tabby: automated gadget chain detection for java deserialization vulnerabilities. In: 2023 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pp. 179–192. IEEE (2023)
22. Dean, J., Grove, D., Chambers, C.: Optimization of object-oriented programs using static class hierarchy analysis. In: Tokoro, M., Pareschi, R. (eds.) ECOOP 1995. LNCS, vol. 952, pp. 77–101. Springer, Heidelberg (1995). https://doi.org/10.1007/3-540-49538-X_5
23. Frohoff, C.: ysoserial (2015). https://github.com/frohoff/ysoserial
24. Grech, N., Smaragdakis, Y.: P/Taint: unified points-to and taint analysis. Proc. ACM Program. Lang. **1**(OOPSLA), 1–28 (2017)
25. Haken, I.: Gadget inspector. https://github.com/JackOfMostTrades/gadgetinspector
26. Hasti, R., et al.: Using static single assignment form to improve flow-insensitive pointer analysis. In: Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation. PLDI '98, New York, NY, USA, pp. 97–105. Association for Computing Machinery (1998)
27. Jordan, H., Scholz, B., Subotić, P.: Soufflé: on synthesis of program analyzers. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016, Part II. LNCS, vol. 9780, pp. 422–430. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41540-6_23
28. Lam, P., et al.: The soot framework for java program analysis: a retrospective. In: Cetus Users and Compiler Infastructure Workshop (CETUS 2011), vol. 15 (2011)
29. Milojkovic, N., et al.: Polymorphism in the spotlight: studying its prevalence in java and smalltalk. In: 2015 IEEE 23rd International Conference on Program Comprehension, pp. 186–195 (2015). https://doi.org/10.1109/ICPC.2015.29
30. Needham, M., Hodler, A.E.: Graph algorithms: practical examples in Apache Spark and Neo4j. O'Reilly Media (2019)
31. Pacheco, C., Ernst, M.D.: Randoop: feedback-directed random testing for java. In: Companion to the 22nd ACM SIGPLAN conference on object-oriented programming systems and applications Companion, pp. 815–816 (2007)
32. Padhye, R., et al.: JQF: coverage-guided property-based testing in java. In: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 398–401. ACM (2019). https://doi.org/10.1145/3293882.3339002
33. Rasheed, S., et al.: A hybrid analysis to detect java serialisation vulnerabilities. In: Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering. ASE '20, pp. 1209–1213. Association for Computing Machinery (2021)
34. Scholz, B., Jordan, H., Subotić, P., Westmann, T.: On fast large-scale program analysis in datalog. In: Proceedings of the 25th International Conference on Compiler Construction, pp. 196–206 (2016)