SENECA: Taint-Based Call Graph Construction for Java Object Deserialization

JOANNA C. S. SANTOS, University of Notre Dame, USA
MEHDI MIRAKHORLI, University of Hawaii at Manoa, USA

Object serialization and deserialization is widely used for storing and preserving objects in files, memory, or database as well as for transporting them across machines, enabling remote interaction among processes and many more. This mechanism relies on reflection, a dynamic language that introduces serious challenges for static analyses. Current state-of-the-art call graph construction algorithms does not fully support object serialization/deserialization, i.e., they are unable to uncover the callback methods that are invoked when objects are serialized and deserialized. Since call graphs are a core data structure for multiple type of analysis (e.g., vulnerability detection), an appropriate analysis cannot be performed since the call graph does not capture hidden (vulnerable) paths that occur via callback methods. In this paper, we present SENECA, an approach for handling serialization with improved soundness in the context of call graph construction. Our approach relies on taint analysis and API modeling to construct sound call graphs. We evaluated our approach with respect to soundness, precision, performance, and usefulness in detecting untrusted object deserialization vulnerabilities. Our results show that SENECA can create sound call graphs with respect to serialization features. The resulting call graphs do not incur significant overhead and was shown to be useful for performing identification of vulnerable paths caused by untrusted object deserialization.

CCS Concepts: • **Software and its engineering → Automated static analysis**; *Software verification and validation.*

Additional Key Words and Phrases: object serialization, taint analysis, call graphs

ACM Reference Format:
Joanna C. S. Santos and Mehdi Mirakhorli. 2024. SENECA: Taint-Based Call Graph Construction for Java Object Deserialization. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 1 (January 2024), 27 pages.

1 Introduction

Static program analysis is a key component of today's software analysis tools that bring automation into activities such as defect localization and/or finding (*e.g.*, [Dolby et al. 2007; Thaller et al. 2020]), vulnerability detection (*e.g.*, [Jovanovic et al. 2006; Liu Ping et al. 2011]), information flow analysis [Sridharan et al. 2011], code refactoring (*e.g.*, [Khatchadourian et al. 2019]), code navigation (*e.g.*, [Feldthaus et al. 2013]), code clone finding (*e.g.*, [Wyrich and Bogner 2019]), and optimization [Hines et al. 2005]. Such tools often perform multiple types of inter-procedural analysis, that leverage ***call graphs*** – data structures that indicate caller-callee relationships [Grove and Chambers 2001]. However, prior works have demonstrated that constructing a call graph for object-oriented programs is often non-trivial, expensive and/or non-feasible due to the usage of many dynamic programming language constructs. For instance, native calls, reflection, and object serialization make it challenging to statically construct a ***sound*** call graph [Ali et al. 2019; Kummita et al. 2021; Reif et al. 2019, 2018; Smaragdakis et al. 2015; Sridharan et al. 2013].

These programming constructs are heavily used in contemporary software systems as they enable the developers to link/load new class libraries, methods, and objects and extend the programs' functionalities [Landman et al. 2017; Reif et al. 2019]. Ignoring such constructs leads to *unsound*

Authors' addresses: Joanna C. S. Santos, Department of Computer Science and Engineering, University of Notre Dame, Notre Dame, IN, 46556, USA, joannacss@nd.edu; Mehdi Mirakhorli, Department of Information and Computer Sciences, University of Hawaii at Manoa, 1680 East-West Road, Honolulu, HI, 96822, USA, mehdi23@hawaii.edu.

2024. 2475-1421/2024/1-ART1 $15.00
https://doi.org/

call graphs in which feasible runtime paths are missed, and call graphs cannot be used to infer the possible execution from the code [Reif et al. 2019, 2018; Sridharan et al. 2013]. To tackle this problem, previous works explored certain classes of language features, such as reflection features [Bodden et al. 2011; Li et al. 2014, 2019; Smaragdakis et al. 2015], native (opaque) code [Smaragdakis et al. 2015], dynamic proxies [Fourtounis et al. 2018], and programs with Remote Method Invocation (RMI) [Sharp and Rountev 2006]. However, as demonstrated by Reif et al. [Reif et al. 2019, 2018], a powerful and frequently used programming construct that has been left out from the programming analysis techniques is *serialization (and deserialization) of objects*.

Object *serialization* is the process of converting (the state of) an object into an abstract representation (*e.g.*, a byte stream or JSON, *etc.*). The reverse process of reconstructing objects from its abstract representation is called *deserialization*. This is a widely used mechanism for storing and preserving objects in files, memory, or database as well as for transporting them across machines, enabling remote interaction among processes and many more. For example, the Android API provides a `Bundle` object which can be used for inter-process communication between apps as well as Android's OS with an individual app via their serialization and deserialization [Arzt et al. 2014; Enck et al. 2014]. Moreover, object (de)serialization is also used to improve the system's performance by saving objects for later retrieval, *e.g.*, saving a trained machine learning model to be used later without the need to retrain the algorithm. Serializing an object has other advantages, such as being readable by applications in other languages. For instance, JavaScript running in a web browser can natively serialize and deserialize objects to and from JSON, therefore interact with other applications written non-JavaScript languages.

Although object serialization is widely used in many languages and commonly adopted by programmers, static analyzers do not fully cover analysis of programs with this construct yet [Reif et al. 2019, 2018]. This is particular important considering the spike of vulnerabilities related to *untrusted object deserialization* [Muñoz and Schneider 2018; Sayar et al. 2023; Schneider and Muñoz 2016] that cannot be automatically detected because call graphs are unsound. For example, Apache's Log4j software library (versions 2.0-beta9 to 2.14.1) had an untrusted object deserialization vulnerability that allowed remote code execution. This was a critical vulnerability that affected several software systems.

As demonstrated by previous studies on the *soundness* of call graph construction approaches [Reif et al. 2019, 2018]— *guaranteeing that all possible behaviors are modeled in a call graph* — state-of-the-art techniques do not support serialization-related operations. They fall shortly in having nodes and edges that represent *callback* methods that are invoked during the serialization or deserialization of objects. There are multiple reasons on why it is hard to handle this language construct:

- Serialization and deserialization uses several overridable callback method(s). These call back methods are invoked by the Java API using "non-trivial" reflective calls that current techniques [Landman et al. 2017] for taming reflection do not address. Therefore, the resulting call graph under-approximate the program's behavior; they miss potential program paths through these call back methods.
- The invoked callbacks during deserialization methods depend on the received object, which is coming from an external stream. The values and internal field types are only known at runtime when the object deserialization occurs.
- The external stream may include objects whose types are not observed statically, *i.e.*, they are available in the classpath (imported libraries, or Java built-in API) but were never actually used (instantiated) in the application scope. A typical static analysis would consider these types as unused.

Therefore, existing techniques on addressing reflections has failed to address call-graph generation with the presence of object serialization/de-serialization [Reif et al. 2019]. As such, potential program flows are disregarded in existing call graph construction algorithms. Since the call graph is a core data structure in performing many inter-procedural code analyses, the underlying client would suffer with the *unsoundness*. In use-cases such as detection of untrusted deserialization vulnerabilities, an appropriate analysis cannot be performed since the call graph does not capture hidden (vulnerable) paths that occur via callback methods. There are two algorithms that (partially) handle serialization constructs (*i.e.*, CHA [Dean et al. 1995] and RTA [Bacon and Sweeney 1996]) but they are imprecise; they abstract program executions to consider more paths than those feasible in the program. Therefore, they *introduce spurious nodes and edges, rendering large call graphs*. Relying on such algorithms for downstream analyses (*e.g.*, vulnerability detection) makes the analysis imprecise, resulting in a high amount of false positives.

A recent line of work [Santos et al. 2021, 2020], presented an approach (named SALSA) for providing support for serialization-related features. Although SALSA aids the static analyses of programs that uses Java's serialization/deserialization API, it is not enough to find hidden (potentially) malicious paths in the program. SALSA relies on API modeling for abstracting the serialization/deserialization protocol which dictates callback methods control and data flow. Specifically, it relies on *downcasts* in the program to infer the callbacks invoked during **deserialization**. However, malicious objects often violate downcasts and are crafted in such way that it triggers the exploit *during* deserialization, *i.e.*, the exploit executes before the downcast is performed [Dietrich et al. 2017a].

Therefore, we introduce in this paper SENECA, a novel approach that handles the challenge of constructing call graphs for programs that uses serialization features. Specifically, we are focusing on improving the call graph's *soundness* for Java programs with respect to serialization and deserialization callbacks without greatly affecting its precision. SENECA performs a ***novel taint-based call graph construction, which relies on the taint state of variables when computing possible dispatches for callback methods***.

The contributions of this work are:

— a novel taint-based call graph construction algorithm to improve *call graphs' soundness with respect to deserialization callbacks*. It is agnostic to the underlying pointer analysis method used to construct a call graph, and it is meant to complement them.
— an evaluation of the approach's soundness, precision, and scalability. Our experiments demonstrated that our approach soundly handled all the six different callbacks that can be invoked during serialization or deserialization.
— a publicly available implementation of SENECA[1].

The rest of this paper is organized as follows: Section 2 describes the serialization and deserialization mechanism and the challenges in creating a call graph that is sound with respect to this feature. Section 3 explains our approach. Subsequently, Section 4 presents the evaluation of the approach whereas Section 5 presents the results. Section 6 contextualizes our approach within the state-of-the art. Section 7 concludes the paper and make final considerations.

2 Background

Multiple programming languages (*e.g.*, Ruby, Python, PHP, and Java) allow objects to be converted into an *abstract representation*, a process called **object serialization** (or "marshalling"). The process

[1]The scripts to reproduce the paper results are currently available on an anonymous repository SenecaRepo. The source code for SENECA will be released upon publication and submitted for artifact evaluation

of reconstructing an object from its underlying abstract representation is called **object deserialization** (or "*unmarshalling*"). Serialization and deserialization of objects are widely used for inter-process communication and for improving the codes' performance by saving objects to be reused later (*e.g.*, saving machine learning models [Ten 2023]).

During object serialization/deserialization, methods from the objects' classes may be invoked. For instance, classes' constructors, getter/setter methods, or methods with specific signatures may be invoked when reconstructing the object. These are the **callback methods** of the serialization/deserialization mechanism. Each programming language has their own serialization/deserialization protocol, abstract representation, and callback methods. The Java's default serialization and deserialization mechanism is thoroughly described at their specification page [Oracle 2010]. We briefly present this mechanism in the next subsection.

2.1 Java Serialization API

The default Java's Serialization API converts a snapshot of an object graph into a *byte stream*. During this process only *data* is serialized (*i.e.*, the object's fields) whereas the code associated with the object's class (*i.e.*, methods) is within the classpath of the receiver [Schneider and Muñoz 2016]. All *non-transient* and *non-static* fields are serialized by default.

The classes `ObjectInputStream` and `ObjectOutputStream` can be used for deserializing and serializing an object, respectively. They can only serialize/deserialize objects whose class implements the `java.io.Serializable` interface. If implemented by a `Serializable` class, the methods listed below can be invoked by Java ***during*** object serialization and/or deserialization:

- **`void writeObject(ObjectOutputStream)`**: it customizes the serialization of the object's state.
- **`Object writeReplace()`**: this method replaces the actual object that will be written in the stream.
- **`void readObject(ObjectInputStream)`**: it customizes the retrieval of an object's state from the stream.
- **`void readObjectNoData()`**: in the exceptional situation that a receiver has a subclass in its classpath but not its super class, this method is invoked to initialize the object's state.
- **`Object readResolve()`**: this is the inverse of `writeResolve`. It allows classes to replace a specific instance that is being read from the stream.
- **`void validateObject()`**: it validates an object after it is deserialized. For this callback to be invoked, the class has to also implement the `ObjectInputValidation` interface and register the validator by invoking the method `registerValidation` from the `ObjectInputStream` class.

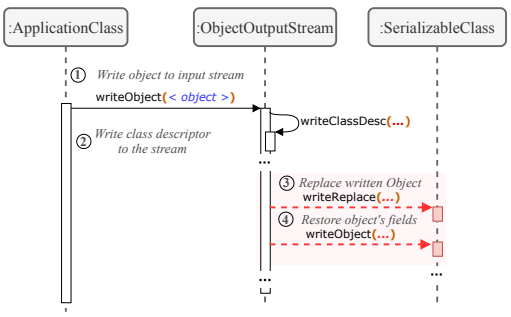

Fig. 1. Callbacks invoked during serialization

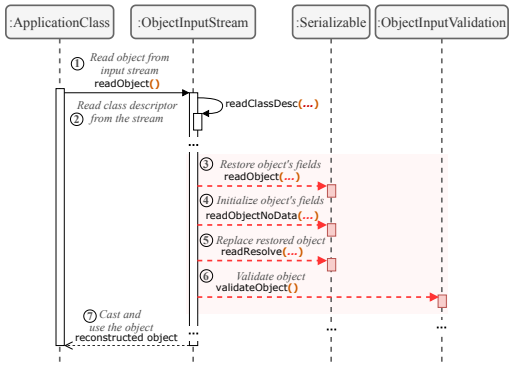

Fig. 2. Callbacks invoked during deserialization

Figures 1 and 2 depicts the sequence of these callback methods invocations. As depicted in this figure, during *serialization* of an object, the callback methods *writeReplace* and *writeObject* are invoked (if these are implemented by the class of the object being deserialized). Similarly, during object *deserializaton*, four callback methods can be invoked, namely, *readObject*, *readObjectNoData*, *readResolve*, and *validateObject*.

2.2 Demonstrative Example

```
1  class Pet implements Serializable {
2    protected String name;
3  }
4  class Cat extends Pet{
5    private void readObject(ObjectInputStream s){
6      /* ... */
7    }
8    private void writeObject(ObjectOutputStream s){
9      /* ... */
10   }
11 }
12 class Dog extends Pet{
13   private Object readResolve(){ /* ... */ }
14   private Object writeReplace(){ /* ... */ }
15 }
16 class Shelter implements Serializable{
17   private List<Pet> pets;
18 }
```

```
19 class SerializationExample{
20   public static void main(String[] args) throws Exception {
21     Shelter s1 = new Shelter(Arrays.asList(new Dog("Max"),
22                                            new Cat("Joy")));
23     File f = new File("pets.txt");
24     FileOutputStream fos = new FileOutputStream(f);
25     ObjectOutputStream out = new ObjectOutputStream(fos);
26     out.writeObject(s1);
27   }
28 }
29 class DeserializationExample{
30   public static void main(String[] args) throws Exception {
31     File f = new File("pets.txt");
32     FileInputStream fs = new FileInputStream(f);
33     ObjectInputStream in = new ObjectInputStream(fs);
34     Shelter s2 = (Shelter) in.readObject();
35   }
36 }
```

Listing 1. Object serialization and deserialization example

Listing 1 has three serializable classes[2]: `Dog`, `Cat` and `Shelter`. Two of these classes have serialization callback methods (lines 5-10 and 13-14). The code at line 21-26 serializes a `Shelter` object `s1` into a file, whose path is provided as program arguments. The code instantiates a `FileOutputStream` and passes the instance to an `ObjectOutputStream`'s constructor during its instantiation. Then, it calls `writeObject(s1)`, which serializes `s1` as a byte stream and saves it into a file. Since the object `s1` has a list field (`pets`) that contains two objects (a `Cat` and a `Dog` instance) the callback methods of these classes invoked.

The `main` method at line 30 deserializes this object from the file. It creates an `ObjectInputStream` instance and invokes the method `readObject()`, which returns an object constructed from the text file. The returned object is casted to the `Shelter` class type. During the deserialization, the methods `readObject` and `readResolve` from the `Cat` and `Dog` classes are invoked, respectively.

2.2.1 Untrusted Object Deserialization To illustrate how a seemingly harmless mechanism can lead to serious vulnerabilities, consider the case that the program in Listing 1 contains two more serializable classes (`CacheManager` and `Task`), as shown in Listing 2. An attacker would create a `CacheManager` object (`cm`) as shown in Figure 3. Then, the attacker serializes and encodes this malicious object (`cm`) into a text file and specifies it as a program argument for the `main` method in Listing 1. When the program reads the object from the file, it triggers the chain of method calls depicted in Figure 3. This sequence of method calls ends in an execution sink (`Runtime.getRuntime.exec()` on line 8 of the `Task` class in Listing 2).

Although this request with a malicious serialized object results in a `ClassCastException`, the malicious command will be executed anyway, because the type cast check occurs ***after*** the

[2]We only show their fields and callback methods due to space constraints.

```java
public class CacheManager implements Serializable {
  private Runnable initHook;
  public CacheManager(Runnable initHook) {
   this.initHook = initHook;
  }
  private void readObject(ObjectInputStream ois) {
    ois.defaultReadObject();
    initHook.run();
  }
}
```

```java
public class Task implements Runnable, Serializable {
  private String command;

  public Task(String command) {
    this.command = command;
  }
  public void run() {
    Runtime.getRuntime().exec(command);
  }
}
```

Listing 2. Gadget classes that can be used to exploit an untrusted object deserialization vulnerablity

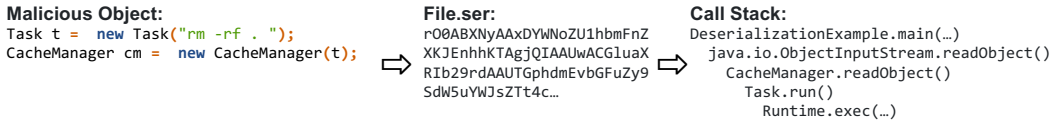

Fig. 3. Malicious serialized object used to trigger a remote code execution

deserialization process took place. As we can see from this example, classes can be specially combined to create a chain of method calls. These classes are called ***"gadget classes"*** as they are used to bootstrap a chain of method calls that will end in an execution sink.

2.3 Challenges for Call Graph Construction

From the examples shown in Section 2.2, we observe two major challenges that should be handled by a static analyzer in order to construct a sound call graph with respect to serialization-related features: (i) the **callback methods** that are invoked during object serialization/deserialization; and (ii) the **fields within the class can be allocated in unexpected ways**, and they dictate which callbacks are invoked at runtime. For instance, if the code snippet in Listing 1 had only the cat object in the list (line 22), then the calls to `readResolve`/`writeReplace` methods in `Dog` would not be made.

Existing pointer analysis algorithms leverage on allocation instructions (*i.e.*, `new T()`) within the program to infer the possible runtime types for objects [Bastani et al. 2019; Feng et al. 2015; Heintze and Tardieu 2001; Hind 2001; Kastrinis and Smaragdakis 2013; Lhoták and Hendren 2006; Rountev et al. 2001; Smaragdakis and Kastrinis 2018]. However, as we demonstrated in the examples, the allocations of objects and their fields and invocations to callback methods are made on-the-fly by Java's serialization/deserialization mechanism. During static analysis, we can only pinpoint that there is an `InputStream` object that provides a stream of bytes from a source (*e.g.*, a file, socket, *etc.*) to an `ObjectInputStream` instance, but the contents of this stream is uncertain. Hence, the deserialized object and its state are unknown (*i.e.*, the allocations within its fields). As a result, existing static analyses fail to support serialization-related features.

3 Seneca: Taint-Based Call Graph Construction for Object Deserialization

To support serialization-related features, Seneca employs an on-the-fly iterative call graph construction technique [Grove et al. 1997], as depicted in Figure 4. It involves two major phases: ① Iterating over a worklist of methods to create the *initial call graph* using an underlying pointer analysis method; ② Refinement of the initial call graph by making a set of assumptions performed iteratively until a fixpoint is reached (*i.e.*, when there are no more methods left in the worklist to be visited).

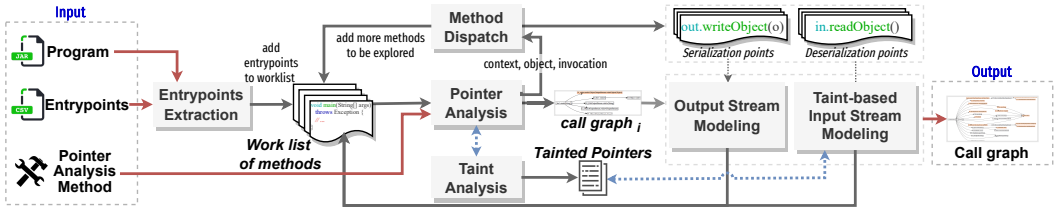

Fig. 4. Our serialization-aware approach for constructing call graphs (SENECA)

3.1 Phase 1: Initial Call Graph Construction

SENECA first takes as input a CSV file with method signatures for the program's **entrypoints** , which are the methods that start the program's execution (*e.g.*, `main()`). The result of this step is a set of entrypoint methods $m \in E$ added to our **worklist** $\mathcal{W}$. This worklist tracks the methods m under a context c that have to be traversed and analyzed, *i.e.*, $\langle m, c\rangle \in \mathcal{W}$, where a **context** c is an abstraction of the program's state. Since the worklist $\mathcal{W}$ tracks methods within a context, the entrypoints methods added to $\mathcal{W}$ are assigned a global context, which we denote as $\emptyset$. Hence, the worklist is initialized as:

$$\mathcal{W} = \{\langle m, \emptyset\rangle \mid \forall m \in E\}$$

Starting from the entrypoint methods identified, SENECA constructs an ***initial (unsound) call graph*** (*i.e.*, call graph$_0$) using the underlying pointer analysis algorithm selected by the client analysis (*e.g.*, `n-CFA`). Each method in the worklist $\langle m, c\rangle \in \mathcal{W}$ is converted into an Intermediary Representation (IR) in Static Single Assignment form (SSA) [Cytron et al. 1991]. Each instruction in this IR is visited following the rules by the underlying pointer analysis algorithm [3]. When analyzing an instance method invocation instruction (*i.e.*, , `x = o.g(`$a_1, a_2, \ldots, a_n$`)`), SENECA computes the possible dispatches (call targets) for the method g as follows: `targets =` $dispatch(pt(\langle o, c\rangle), g)$. This dispatch mechanism takes into account the current points-to set for the object o at the current context c as well as the declared target g. If the invocation instruction occurs at a **serialization** or **deserialization point** , then the *dispatch* function implemented by our approach creates a **synthetic method** to model the runtime behavior for the `readObject()` and `writeObject()` from the classes `ObjectInputStream` and `ObjectOutputStream`, respectively.

These synthetic models are initially created *without* instructions. Their instructions are constructed during the call graph refinement phase (Phase 2). It is important to highlight that the calls to synthetic methods (models) are *1-callsite-sensitive* [Sridharan et al. 2013]. We use this context-sensitiveness policy to account for the fact that one can use the same `ObjectInputStream/ObjectOutputStream` instance to read/write multiple objects. Thus, we want to disambiguate these paths in the call graph.

As a result of this first iteration over Phase 1, we obtain the **initial call graph** (g_0) and a **list of the call sites at the serialization and deserialization points**.

3.2 Phase 2: Call Graph Refinement

In this phase, we take as input the current call graph g_i which contains as nodes actual methods in the application and synthetic methods created by our approach in the previous phase.

[3]We point the reader to the work by Sridharan *et al.* [Sridharan et al. 2013] which provides a generic formulation for multiple points-to analysis policies.

3.2.1 Object Serialization Abstraction Algorithm 1 indicates the procedure for modeling object serialization. For each instruction at the serialization points, we obtain the points-to set for the object o_i passed as the first argument to `writeObject(Object)`. The points-to set $pt(\langle o_i, c\rangle)$ indicates the set of allocated types t for o_i under context c. Since the writeObject's argument is of type Object, we first add to m_s a type cast instruction that refines the first parameter to the type t. In case the class type t implements the `writeObject(ObjectInputStream)` callback, we add an invocation instruction from m_s targeting this callback method.

Subsequently, we iterate over all non-static fields f from the class t and compute their points-to sets (see the **foreach** in line 10). If the concrete types allocated to the field contains callback methods, we add three instructions: (i) an instruction to get the instance field f from the object; (ii) a downcast to the field's type; (iii) an invocation to the callback method from the field's declaring class.

It is important to highlight the edge case scenario when the object being serialized is a *java.util.Collection* or a *java.util.Map*. In this case, Seneca tracks what objects were *added* to the collection in order to add invocations to their callback methods (if provided).

After adding all the needed instructions to the synthetic method m_s, we re-add the synthetic method to Seneca's worklist (as depicted in Figure 4).

Algorithm 1: Object serialization modeling

```
   Input: Set of invocation instructions to writeObject: I;
          Project's initial call graph: G;
   Output: Set of refined synthetic models M_s
1  foreach instruction in I do
2      o_i ← argument(1,instruction)
3      c ← context(instruction)
4      m_s ← target(instruction)
5      foreach t ∈ pt(⟨o_i, c⟩) do
6          addTypeCast(m_s,t)
7          if t has a callback method then
8              addInvoke(m_s, t.callback)
9          end
10         foreach f ∈ fields(t) do
11             foreach fieldType ∈ pt(⟨o_i.f, c⟩) do
12                 if fieldType has callback then
13                     addGetField(m_s, f)
14                     addTypeCast(m_s, fieldType)
15                     addInvoke(m_s, fieldType.callback)
16                 end
17             end
18         end
19     end
20     addToWorkList(m_s,c)
21 end
```

3.2.2 Taint-Based Object Deserialization Abstraction Starting from the ***deserialization points*** identified, Seneca computes the call graph on-the-fly by iteratively solving constraints over the instructions. Each method in the worklist $\langle m, c\rangle \in \mathcal{W}$ is converted into an Intermediary Representation (IR) in Single Static Assignment form (SSA) [Cytron et al. 1991; Rosen et al. 1988]. Moreover, these methods have special variables to denote their return value m_{ret} and the `this` pointer $m.this$ (for non-static methods).

For each method in the worklist $\mathcal{W}$, Seneca performs *pointer analysis* in parallel with *taint analysis* to compute the taint state of variables and points-to sets. Each instruction in the method's IR is visited following the rules by the underlying pointer analysis [Sridharan et al. 2013] and our

Table 1. Taint propagation rules employed by SENECA when building call graphs.

Instruction at method m in a context c	Taint propagation Rule	
`x = T.f`	$\tau(x) = \tau(x) \vee \tau(T.f)$	[Load-Static]
`x = y.f`	$\tau(x) = \tau(x) \vee \tau(y) \vee \tau(y.f)$	[Load-Instance]
`x.f = y`	$\tau(x.f) = \tau(x.f) \vee \tau(y)$	[Store-Instance]
`T.f = y`	$\tau(T.f) = \tau(T.f) \vee \tau(y)$	[Store-Static]
`x = o.g(a1,···,an)`	$\forall a_i \in A_j, \forall p_i \in P_g : \tau(p_i) = \tau(p_i) \vee \tau(a_i), \tau(g_{this}) = \tau(g_{this}) \vee \tau(o)$	[Instance-Call-Args]
	$\tau(x) = \tau(x) \vee \tau(g_{ret})$	[Instance-Call-Return]
	Side Effect: $\tau(o) = true \rightarrow pt(\langle o, c\rangle) = pt(\langle o, c\rangle) \cup targetTypes(o, c, g)$	[Call-Side-Effect]
`x = T.g(a1,···,an)`	$\forall a_i \in A_j, \forall p_i \in P_g : \tau(p_i) = \tau(p_i) \vee \tau(a_i)$	[Static-Call-Args]
	$\tau(x) = \tau(x) \vee \tau(g_{ret})$	[Static-Call-Return]
`return x`	$\tau(m_{ret}) = \tau(m_{ret}) \vee \tau(x)$	[Return]
	Side Effect: $\mathcal{W} = \mathcal{W} \cup C_m$	[Return-Side-Effect]
`x = y[i]`	$\tau(x) = \tau(x) \vee \tau(y)$	[Array-Load]
`x[i] = y`	$\tau(x) = \tau(x) \vee \tau(y)$	[Array-Store]
`ϕ = v1,v2,···,vn`	$\tau(\phi) = \tau(v_1) \vee \tau(v_2) \vee \ldots \vee \tau(v_n)$	[Phi]
`x = (TypeCast) y`	$\tau(x) = \tau(x) \vee \tau(y)$	[Checkcast]

taint analysis algorithm. Thus, each pointer in a program has an associated taint state $\tau(p)$, where $\tau(p) = true$ denotes a tainted pointer and $\tau(p) = false$ denotes an untainted (safe) pointer. Below, we provide the formulation of our taint analysis policy [Schwartz et al. 2010].

► **Taint Introduction:** As described before, deserialization points are replaced by a *synthetic method*, *i.e.*, a "fake call graph node" [Sridharan et al. 2013]. It is a synthetic method created on-the-fly to model: (i) the instantiation of the class G_c that contains a callback method(s) m_c; (ii) the invocation to the callback method(s) using the newly created object; and (iii) the instantiation of any parameters for the magic methods. It is important to highlight that in the Step (i), when instantiating the callback method's object, we invoke the class' default constructor. This is to follow the Java's deserialization process (see Section 2).

Therefore, SENECA initializes the following pointers as tainted:

- The pointer for `x` in the instruction `x = new` G_c`()`, where G_c denotes a class that contains a deserialization callback method (*e.g.*, `readResolve`):

$$\tau(x) = \texttt{true}$$

- The pointers for all the fields of `x`:

$$\forall f_i \in fields(x) : \tau(x.f_i) = \texttt{true}$$

- The `this` pointer in the callback method m_c that is invoked:

$$\tau(m_c.this) = \texttt{true}.$$

► **Taint Propagation Rules:**

As the method's instructions are parsed, we employ the rules listed in Table 1 to compute the taint states of the program's variables. As shown in Table 1, the rules for assignment instructions are as follows:

$$\texttt{lhs = rhs} \longrightarrow \tau(lhs) = \tau(lhs) \vee \tau(rhs)$$

That is, the pointer for the left-hand side is tainted if the pointer for the right-hand side is also tainted (or the left-hand side itself was already previously tainted). This is the case for the rules LOAD-STATIC, LOAD-INSTANCE, STORE-INSTANCE, STORE-STATIC, STATIC-CALL-RETURN, RETURN, ARRAY-LOAD, ARRAY-STORE, and CHECKCAST.

Phi functions (ϕ) are special statements that are inserted into a method's SSA form to represent possible values for a variable depending on the control flow path taken. The taint for the pointer of phi $\tau(\phi)$ will be tainted if *any* of the possible variables' pointers are tainted.

When there is a method invocation, it can either be a static invocation or an invocation to a instance method. In both cases, each passed parameter p_i is assigned to the corresponding argument a_i from the invoked method. Consequently, the rules Instance-Call-Args, and Static-Call-Args are propagated likewise assignment instructions. Notice, however, that for instance methods there is a special variable m_{this} denoting the "this" pointer for that method. Hence, the rule Instance-Call-Args propagates the taint from the caller object to the "this" pointer $\tau(g_{this})$.

It is worth to highlight that taint ***is never removed*** from a pointer. Although this will make the underlying call graph more imprecise, our goal is to soundly reason over *all* possible runtime paths.

–**Side Effects to the Pointer Analysis Engine**:

Method invocations and return instructions introduce *side-effects* to the static analysis engine state, labelled in Table 1 as Call-Side-Effect and Return-Side-Effect, respectively.

- *Instance method invocations*: When there is an instance method invocation $o.g(...)$ and the object o is tainted, then Seneca computes the possible method targets for the call $o.g(...)$ *soundly*. The dispatch is computed as described below:

(1) it obtains the static type t for o, i.e. $t = type(o)$;
(2) it extracts the set of classes based on the inheritance hierarchy for T (i.e., $T = cone(t)$, where $cone(t)$ returns the list of all descendants of t, including t itself [Tip and Palsberg 2000]).
(3) it computes the subset $C \subseteq T$ that includes only the types (classes) which provide a concrete implementation matching the signature of the invoked method g.
(4) it computes the subset $A_t \subseteq C$ which include only classes that are accessible to t according to Java's visibility rules[4].
(5) finally, the possible target methods are all the methods from the set A_t in which their classes are serializable (i.e., implements the serializable interface directly or via inheritance).

As one can notice, this dispatch is similar to the one employed by Class Hierarchy Analysis (CHA). The main difference are in steps (4) and (5), where Seneca takes into account class visibility rules as well as whether the type is serializable.

Once the dispatch is computed (`targetTypes(o,g)` in Call-Side-Effect) the points to set for $pt(\langle o, c\rangle)$ adds all the elements from $targetTypes(o, g)$.

- *Method return values*: In a scenario where a method m has a tainted return value $\tau(m_{ret}) = \texttt{true}$, all the callers of m are re-added to the $\mathcal{W}$. Since the return is tainted, we need to back propagate this information to all the callers of m to ensure that the rules Instance-Call-Return and Static-Call-Return are applied correctly.

— **Context-sensitivity for Tainted Method Calls** Our taint-based call graph construction algorithm is ***agnostic*** to the pointer analysis policy (*e.g.*, 0-1-CFA). This means that a client analysis could choose to use a context insensitive analysis (*e.g.*, 0-CFA). Since tainted pointers are likely to have a large points-to set because we use a sound analysis to compute all possibilities for method dispatches when the receiver object is tainted, we should avoid merging point-to-sets of these tainted variables. Otherwise, the resulting pointer analysis would be too imprecise to be used by downstream client analyses.

Therefore, we use ***1-callsite-sensitivity*** for tainted method calls (even if we use an insensitive analysis for all the other pointers).

[4]Visibility rules are thoroughly described in the language specification https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html#jvms-4.1-200-E.1

► **Demonstrative Example:** Consider the code snippet in Listing 3. The class `Main` has a `main` method that reads an object from a file, whose path is provided as a program argument. This program contains other four classes (`CacheManager`, `TaskExecutor`, `CommandTask`, and `Config`). We demonstrate SENECA's taint-based deserialization modeling considering that we selected `0-1-CFA` as the main pointer analysis method.

```
1  class Main {
2    public static void main(String[] a)
3        throws Exception {
4      FileInputStream f=new FileInputStream(a[0]);
5      ObjectInputStream in=new ObjectInputStream(f);
6      Config obj = (Config) in.readObject();
7    }
8  }
9  class CommandTask
10     implements Runnable, Serializable {
11   private String cmd;
12   private TaskExecutor taskExecutor;
13   @Override
14   public void run() {
15     if (!cmd.isEmpty() && taskExecutor != null)
16       taskExecutor.executeCmd(cmd); /* site @24 */
17   }
18 }
19 class TaskExecutor implements Serializable {
20   public void executeCmd(String cmd) {
21     try {
22       Runtime rt = Runtime.getRuntime();
23       rt.exec(cmd);
24     } catch (IOException e) {  }
25   }
26 }
27 class Config implements Serializable {
28   private String page;
29   public void readObject(ObjectInputStream ois)
30       throws IOException, ClassNotFoundException {
31     ois.defaultReadObject();
32     Runtime rt = Runtime.getRuntime();
33     rt.exec("open http://localhost/" + page);
34   }
35 }
```

```
1  class CacheManager implements Serializable {
2    private Runnable task;
3    private Runnable[] taskArray;
4    private List<Runnable> taskList;
5    private Set<Runnable> taskSet;
6    private Map<String, Runnable> taskMap;
7    private String os;
8    private long timestamp;
9    public void readObject(ObjectInputStream ois)
10       throws IOException, ClassNotFoundException {
11     ois.defaultReadObject();
12     Runnable r;
13     if(os.equals("windows") && task instanceof CommandTask){
14       r = getInitHook();     /* site @32 */
15       r.run();
16     }else {
17       r = getFromArray();
18       r.run();               /* site @46 */
19       r = getFromList();
20       r.run();               /* site @57 */
21       r = getFromSet();
22       r.run();               /* site @68 */
23       r = getFromMap();
24       r.run();               /* site @79 */
25     }
26   }
27   Runnable getInitHook(){ return task; }
28   Runnable getFromArray() { return taskArray[0]; }
29   Runnable getFromList() { return taskList.get(0); }
30   Runnable getFromSet() { return taskSet.iterator().next(); }
31   Runnable getFromMap() { return taskMap.get("xyz"); }
32 }
```

Listing 3. Walk-through example to demonstrate SENECA's approach

SENECA first extracts the program's entrypoints, provided as part of the analysis configuration. In this example, the `Main.main(String a[])` is specified as the main method. Therefore, the SENECA's worklist is initialized as: $\mathcal{W} = \{\langle Main.main(String\ a[]), \emptyset\rangle\}$. SENECA then proceeds to iteratively compute the call graph by traversing each instruction for each method in the worklist.

There are three method invocations on `Main.main()`: two invocations to the constructors (`<init>`) of `FileInputStream` and `ObjectInputStream` classes followed by a call to the `readObject()` method from the `ObjectInputStream` class. The invocation to `ObjectInputStream.readObject()` is replaced by SENECA with a model (synthetic) method that has the same signature, but it is initialized without any instructions. At this stage, the call graph for this program after traversing the main method looks like as shown in Listing 5. All these three call graph nodes discovered after parsing `Main.main()` are added to the worklist to be processed (*i.e.*, , `FileInputStream.<init>()`, `ObjectInputStream.<init>()`, and `ObjectInputStream.readObject()`.

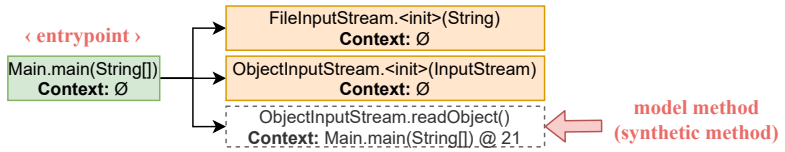

Fig. 5. Initial call graph after parsing the `Main.main()` method in Listing 3

The instructions that are added to `ObjectInputStream.readObject()` relies on taint states to infer callback methods that might by invoked during deserialization. Thus, when refining a method model, SENECA considers that *all* serializable classes in the classpath could have its callbacks invoked. By using this strategy, there are two possible callbacks that can be invoked: one from `Config` and one from `CacheManager`. Hence, all of its instance fields are marked as *tainted* per the taint introduction rules previous described (these are highlighted in red on Listing 3). Based on the taint propagation rules specified on Listing 1, variables are then marked as tainted (these variables that are tainted due to propagation are highlighted in cyan on Listing 3).

Recall that tainted invocations (*i.e.*, an instruction such as `obj.aMethod()` in which `obj` is tainted) are handled differently. Whereas the dispatch of non-tainted invocation will follow the rules from the underlying pointer analysis policy, the dispatch for tainted invocations are computed using a modified version of the CHA algorithm. Therefore, the computed call graph when using the taint-based approach looks like as Listing 6[5]. As shown in this image, the model method includes the following instructions: an object instantiation for `Config` as well as `CacheManager`, their constructors invocation, and invocations to their callback methods. Finally, the model method returns a value that can either be an instance of `Config` or `CacheManager`. Notice that the phi function (ϕ) added to indicate this possibility.

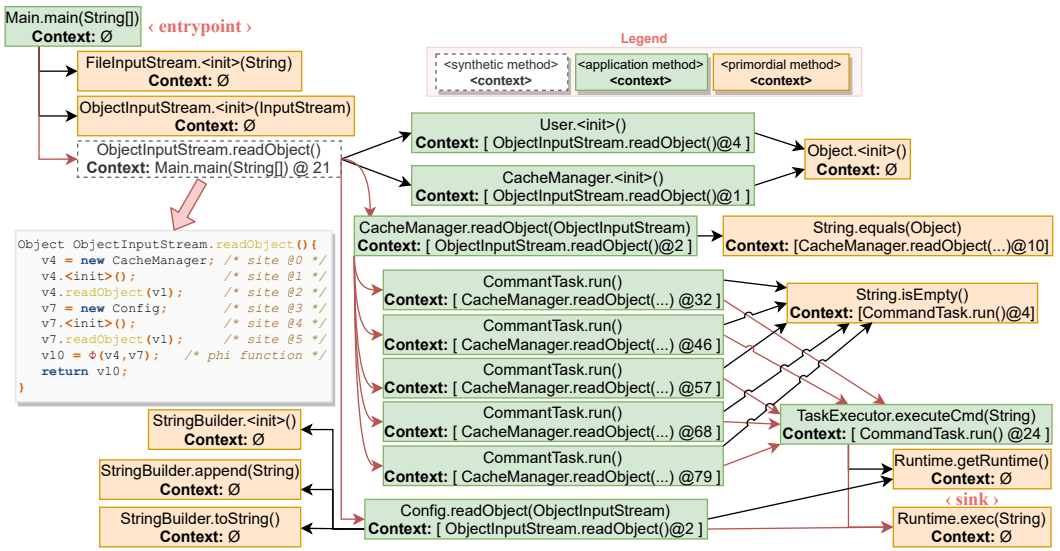

Fig. 6. Call graph for Listing 3 when using the taint-based strategy

[5]Due to space constraints, we elide the "getter" calls as well as inner calls from primordial nodes (*e.g.*, `String.isEmpty()`)

4 Evaluation

In this section, we introduce our research questions and describe our experiment setup and design to answer those.

4.1 Research Questions

This paper addresses the following research questions:

RQ1 Soundness. *Does Seneca handle object deserialization soundly?*
RQ2 Precision. *Does an increase in the call graph's soundness incur a significant loss in its precision?*
RQ3 Scalability. *Does Seneca scale well for real software systems?*
RQ4 Usefulness. *Is Seneca useful for a client analysis focused on vulnerability detection?*

To answer the aforementioned research questions, we developed a prototype for Seneca in Java using IBM's T. J. Watson Libraries for Analysis (**WALA**) [IBM [n.d.]]. It allows client analyses to select a pointer analysis method that can either be `0-n-CFA`, or `n-CFA`, where *n* is provided. We explain in the next subsections the methodology and datasets used to answer each RQ.

4.2 Answering RQ1: Soundness

We aim to verify whether Seneca improves a call graph's ***soundness*** with respect to deserialization callbacks and how it compares with existing approaches [Reif et al. 2019, 2018; Santos et al. 2021, 2020]. The soundness of a call graph construction algorithm corresponds to being able to create a call graph that incorporate ***all*** possible paths (nodes and edges) that can arise at runtime [Ali et al. 2019; Kummita et al. 2021]. In this work, we are specifically looking at improving a call graph's soundness to cover possible invocations that arise **during object serialization and deserialization**. Therefore, we use two datasets to answer this first research question:

Table 2. Test cases from the CATS Test Suite [Eichberg 2020] and which soundness aspect they aim to verify.

ID	Description
Ser1	The code serializes an object whose class contains a custom `writeObject` method. It tests whether the call graph creates a node for the `writeObject(...)` callback method that can be invoked by the writeObject method from the ObjectOutputStream class.
Ser2	Tests whether the call graph has nodes and edges for the `writeObject` callback method under the scenario that the call *may* be invoked if a condition is true.
Ser3	Tests whether the call graph construction algorithm considers inter-procedural flow to soundly infer that the object's `writeObject(...)` callback method will be invoked by the writeObject method from the ObjectOutputStream class.
Ser4	The code deserializes an object (without performing a downcast) whose class contains a custom `readObject` method. It tests whether the call graph creates a node for the `readObject(...)` callback method that can be invoked by the readObject method from the ObjectInputStream class.
Ser5	The code deserializes an object whose class contains a custom `redObject` method. It tests whether the call graph creates a node for the `readObject(...)` callback method that can be invoked by the readObject method from the ObjectInputStream class. Unlike Ser4, this test case has a downcast to the expected type of the read object.
Ser6	Tests whether the call graph has nodes and edges for the `writeReplace` callback method that will be invoked during serialization.
Ser7	Tests whether the call graph has nodes and edges for the `readResolve` callback method that will be invoked during deserialization.
Ser8	Tests whether the call graph has nodes and edges for the `validateObject` callback method that will be invoked during deserialization.
Ser9	Tests whether constructors of serializable classes are handled soundly. It checks whether the call graph models the runtime behavior, which invokes the first default constructor that is not from a serializable superclass.

- **Call Graph Assessment & Test Suite (CATS) [Eichberg 2020]**: This dataset was released as part of recent empirical studies [Reif et al. 2019, 2018] to investigate the soundness of the call graphs computed by existing algorithms with respect to particular programming language constructs. The CATS test suite[6] was derived by an extensive analysis of real Java projects to create test cases that are representative of common ways that projects use these language

[6]This project was formerly known as the Java Call Graph Test Suite (JCG).

constructs (*e.g.*, lambdas, reflection, serialization, etc.). The dataset includes **9** test cases for verifying the soundness of call graphs during serialization and deserialization of objects. Each test case is a Java program with annotations that indicate the expected target for a given method call. Table 2 provides an overview of the test cases available in the CATS test suit and what aspects they aim to probe. Hence, in this first experiment, we run Seneca using two pointer analysis configurations: `0-1-CFA`, and `1-CFA`. Then, we compare it against Salsa (`0-CFA`, `1-CFA`), a state-of-the-art tool, as well as the same algorithms used in the empirical study by Reif *et al.* [Reif et al. 2019], namely Soot (`CHA`, `RTA`, `VTA`, and `Spark`), Wala (`RTA`, `0-CFA`, `1-CFA`, and `0-1-CFA`), Doop (`context-insensitive`), and Opal (`RTA`).
— ***Metric***: Likewise to the prior empirical study by Reif *et al.* [Reif et al. 2019, 2018], we compute the number of *failed* and *passed* test cases for each approach as a way to investigate the soundness of our approach.

- **XCorpus dataset**: Although the CATS dataset was carefully constructed to test call graph construction algorithms with respect to programming language features, the test cases are small programs (*i.e.*, with few serializable classes). Therefore, to enhance our analysis, we used programs available on the XCorpus dataset [Dietrich et al. 2017b]. We chose this dataset because it has been widely used in prior related works [Fourtounis et al. 2018; Santos et al. 2021, 2020] and it was manually curated to be representative of real Java projects. From this dataset, we selected a total of 10 programs from the XCorpus dataset [Dietrich et al. 2017b] (listed in Table 3). We selected these projects because they match the following criteria: (i) they perform object serialization / deserialization; (ii) they contain serializable classes that provide custom implementation for callback methods; hence, they would be suitable to verify whether our approach can properly compute a call graph that uncover hidden paths via callback methods.
  For each of these 10 projects, we created a set of test cases that exercised the serialization and deserialization of objects from the classes that contained custom callback methods. Each test case serializes an object into a file, and then deserializes it back from this file, as shown in Listing 4.

```
public class TC<number> {
    private static Object getObject() {
        Object object = <initialization>
        return object;
    }
    public static void main(String[] args) throws Exception {
        Object obj = getObject();
        FileOutputStream fOut = new FileOutputStream(args[0]);
        ObjectOutputStream objOut = new ObjectOutputStream(fOut);
        objOut.writeObject(obj);
        FileInputStream fs = new FileInputStream(args[0]);
        ObjectInputStream objIn = new ObjectInputStream(fs);
        Object deserializedObj = objIn.readObject();
        new File(filepath).delete();
    }
}
```

Listing 4. Test Case template

The systematic process we followed to create these test cases were as follows. For each class in the XCorpus program that had a custom callback method, we created a "simple" test case. This "simple" test case returns a single instance from the class inside the method *getObject()*. We read the project's documentation to initialize the object's fields correctly and avoid exceptions thrown by the class' constructor. We also created "composite" test cases in which the class instance is wrapped into a collection, *i.e.*, an *ArrayList*, a *HashSet*, a *HashMap*, or an *array*.

By following this systematic process, we create five test cases (1 "simple" test case, and 4 "composite" ones) for each class with a custom callback in an XCorpus project. The list of XCorpus programs and the number of test cases for each of them is shown in Table 3.

After creating these test cases, we execute them to extract their **dynamic call graph** (runtime call graph). We implemented a JVMTI (Java Virtual Machine Tool Interface) agent in C to compute these runtime call graphs. This implementation has an instrumentation agent that is attached to the program's execution. It captures every method that is invoked in the program and its caller method.

Since we aim to investigate whether our taint-based call graph algorithm handle object deserialization soundly or would unsound assumptions be able to find vulnerabilities, we compare Seneca against Salsa [Santos et al. 2021, 2020], a state-of-the-art tool that computes call graphs for object deserialization based on downcasts within the program, which yields to less sound call graphs.

Metric: Similar to prior works [Ali et al. 2019; Ali and Lhoták 2012; Kummita et al. 2021; Li et al. 2014; Smaragdakis et al. 2015], we verify our approach's soundness based on the number of edges in the runtime call graph that are *missing* in the static call graph. In our comparison, we differentiate *application-to-application* edges, *application-to-library*, and *library-to-library* edges. That means that we disregard missing edges due to: (a) *class initializers* (because `<clinit>` methods are modeled by Wala using a synthetic method that invokes all class initializers at once), (b) *native code* (because it cannot be statically analyzed), (c) *explicitly excluded classes* (*i.e.*, classes inside our list of exclusions file that are removed from the static call graph), and (d) *library-to-library* edges (*i.e.*, edges from a built-in Java class to another built-in language class).

Table 3. XCorpus programs [Dietrich et al. 2017b] used in our experiments and the number of Test Cases (TCs) created for each

Project	batik (1.7)	castor (1.3.1)	james (2.2.0)	jgraph (5.13.0.0)	jpf (1.5.1)	log4j (1.2.16)	openjms (0.7.7-beta-1)	pooka (3.0-080505)	xalan (2.7.1)	xerces (2.10.0)
# Classes	2,560	1,639	340	187	152	308	808	1,617	1,621	1,034
# Classes in Dependencies	1,209	947	274	0	1	0	28	0	0	0
# Test Cases	25	65	5	30	5	15	5	30	25	5

4.3 Answering RQ2: Precision

Although soundness is a desirable property for static analysis, in practice, however, creating a sound analysis also implies a loss of precision. Due to the undecidability of program verification, it is impossible to create an analysis that is both *sound* and *precise* [Rice 1953]. Therefore, a sound analysis is an over-approximation that may include spurious results (*e.g.*, unrealistic paths).

While our approach aims to enhance an existing call graph construction algorithm to handle serialization-related callbacks soundly, we need to verify whether our approach introduces imprecision and to what extent. Imprecision in this work refers to adding nodes and edges that will not arise at the program's runtime during object serialization and deserialization [Ali et al. 2019]. In other terms, the precision of a call graph means that it does not contain nodes and edges that will not arise at runtime during object deserialization and serialization.

To answer this question, we use our JVMTI agent to compute the runtime call graph for each program in the CATS test suite [Eichberg 2020] and our manually constructed Test Cases derived

from the XCorpus dataset [Dietrich et al. 2017b]. Subsequently, we compute the number of edges in the *static* call graph that did not exist in the *runtime* call graph.

— ***Metric***: We calculate the number of nodes and edges that appeared in SENECA's call graph but did not appear on the dynamic call graph. Similar to prior works [Smaragdakis et al. 2015; Smaragdakis and Kastrinis 2018], when performing this calculation, we only consider *application-to-application* edges and *application-to-library* edges as long as these edges do not include nodes that are a *class initializer*, a *native code method*, or a method from an *explicitly excluded class*.

4.4 Answering RQ3: Performance

Our serialization-aware call graph construction approach introduces *extra iterations* on the underlying pointer analysis methods. As such, we investigate whether these extra iterations introduce significant overhead that renders the analysis impractical for real large-scale programs.

To verify the overhead of incurred by our approach, we first use SENECA to build the call graphs for the test cases created for the 10 programs extracted from the XCorpus dataset [Dietrich et al. 2017b]. Subsequently, we run the `0-1-CFA` and `1-CFA` call graph construction algorithms available in WALA ***with*** and ***without*** our serialization-aware approach enabled. For comparison, we also ran SALSA configured with `0-1-CFA` and `1-CFA` to build call graphs. For all of these approaches, we used a standard list of class exclusions [7]; these classes are ignored during call graph construction by WALA, SALSA, and SENECA.

— ***Metric***: We measure **(i)** the *running time* to compute the call graphs when using our approach, and **(ii)** the extra *added number of iterations* over the worklist of the call graph construction algorithm. We run these analyses on a machine with a 2.9 GHz Intel Core i7 processor and 32 Gb of RAM memory.

4.5 Answering RQ4: Efficiency

One of the premises of this work is that a taint-based call graph construction enable the computation of sound call graphs with respect to (de)serialization, which can be useful for client analyses, such as vulnerability detection. In this question, we aim to verify whether SENECA can help a static analysis technique in finding potential vulnerable paths in the program.

To answer this question, we obtained 3 open-source projects with known disclosed deserialization vulnerabilities. We selected these projects because their exploits have been widely discussed by practitioners and are available on the YSoSerial GitHub repository [Frohoff 2018]. That is, these projects have well-known "gadget chains" which were previously disclosed in vulnerability reports.

To answer this RQ, we used SENECA and SALSA to compute the call graphs of these projects (configured to use `0-1-CFA` and `1-CFA`). Subsequently, we use these call graphs to extract vulnerable paths which are paths from `ObjectInputStream.readObject()` to *sinks*, *i.e.*, method invocations to security-sensitive operations.

To identify sinks, we manually curated a list of security-sensitive method signatures. To do so, we extracted the list of sink methods from a prior published work [Thomé et al. 2017]. Moreover, we parsed the manifest file from the Juliet Test Suite [Software 2023]. This test suite is a dataset from NIST (National Institute of Standards and Technology) which has a collection of synthetic C/C++ and Java code samples with different software weaknesses (CWEs). Their manifest file indicates all the files for a test case, the kind of weakness it contains, and its location in the code. Thus, we parsed the manifest to extract the lines that are flagged as vulnerable, filtered out the lines that are not method invocations, grouped them by signature, and manually identified the ones that

[7] https://github.com/wala/WALA/blob/master/com.ibm.wala.core/src/main/resources/Java60RegressionExclusions.txt

are sinks. After performing these two complementary curation steps, we obtained a total of **101** methods signatures for sinks.

— ***Metric:*** We measured how many *vulnerable paths* each approach was able to identify.

5 Results

5.1 RQ1: Call Graph Soundness

This section describes the results of the experiments for measuring the soundness of the call graphs computed by Seneca.

5.1.1 Dataset #1: CATS Table 4 reports the programs in which each approach soundly inferred the call graph (✓) and the ones it failed to do so (✗). As shown in this table, we built call graphs using two different pointer analysis policies: `0-1-CFA`, and `1-CFA`. For the sake of comparison, this table also includes the same algorithms and results presented by Reif *et al.* [Reif et al. 2019] and that we were able to reproduce using the Docker image [Reif 2023] provided by their work. The released artifacts of Reif *et al.* study [Reif et al. 2019] includes adapters for constructing call graphs using Soot (`CHA`, `RTA`, `VTA`, and `Spark`), Wala (`RTA`, `0-CFA`, `1-CFA`, and `0-1-CFA`), Doop (`context-insensitive`), and Opal (`RTA`). We also included a comparison with a recent published work, Salsa [Santos et al. 2021, 2020], configured with the same pointer analysis policies as ours, *i.e.*, `0-1-CFA`, and `1-CFA`.

Table 4. Results from running the test cases from CATS

	Salsa (0-1-CFA)	Salsa (1-CFA)	Seneca (0-1-CFA)	Seneca (1-CFA)	Opal (RTA)	Soot (CHA)	Soot (RTA)	Soot (Spark)	Soot (VTA)	Wala (0-1-CFA)	Wala (0-CFA)	Wala (1-CFA)	Wala (RTA)
Ser1	✓	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗
Ser2	✓	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗
Ser3	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗
Ser4	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗
Ser5	✓	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗
Ser6	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗
Ser7	✓	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗
Ser8	✓	✓	✓	✓	✗	✓	✓	✗	✗	✗	✗	✗	✗
Ser9	✓	✓	✓	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗

As shown in Table 4, only our serialization-aware call graph construction (Seneca) and Salsa passed ***all*** of the nine test cases. Only three other algorithms partially provided support for callback methods, namely Soot_{RTA} and Soot_{CHA} (2 out of 9) and OPAL_{RTA} (5 out of 9) [Reif et al. 2019]. The remaining algorithms, *i.e.*, Soot (VTA, and Spark), Wala (RTA, 0-CFA, 1-CFA, 0-1-CFA), and Doop (context-insensitive), did not provide support at all for serialization-related callback methods.

It is also important to highlight that the frameworks that provided partial support for serialization-related features (Soot_{RTA}, Soot_{CHA}, and Opal_{RTA}) use ***imprecise*** call graph construction algorithms (CHA [Dean et al. 1995] or RTA [Bacon and Sweeney 1996]). Table 5 shows a comparison of call graphs' sizes in terms of nodes and edges. As we can infer from these charts, the only call graph construction algorithms used by Soot, and Opal that provided **partial** support for serialization create much larger call graphs (in terms of the number of nodes and edges). Since these algorithms only rely on static types when computing the possible targets of a method invocation, they introduce spurious nodes and edges, thereby increasing the call graph's size.

Our approach enhances the underlying pointer analysis policy in order to strike a balance between improving soundness while not greatly affecting the call graph's precision by adding

Table 5. Call Graph sizes for each approach

TC	Approach	# Nodes	# Edges
Ser1	OPAL$_{RTA}$	5,983	39,580
	Salsa$_{0\text{-}1\text{-}CFA}$	771	1,527
	Salsa$_{1\text{-}CFA}$	1,876	3,538
	Seneca$_{0\text{-}1\text{-}CFA}$	771	1,527
	Seneca$_{1\text{-}CFA}$	1,876	3,538
Ser2	OPAL$_{RTA}$	5,985	39,583
	Salsa$_{0\text{-}1\text{-}CFA}$	772	1,529
	Salsa$_{1\text{-}CFA}$	1,878	3,540
	Seneca$_{0\text{-}1\text{-}CFA}$	772	1,529
	Seneca$_{1\text{-}CFA}$	1,878	3,540
Ser3	Salsa$_{0\text{-}1\text{-}CFA}$	772	1,528
	Salsa$_{1\text{-}CFA}$	1,877	3,539
	Seneca$_{0\text{-}1\text{-}CFA}$	772	1,528
	Seneca$_{1\text{-}CFA}$	1,877	3,539
Ser4	Salsa$_{0\text{-}1\text{-}CFA}$	722	1,323
	Salsa$_{1\text{-}CFA}$	1,590	2,841
Ser4	Seneca$_{0\text{-}1\text{-}CFA}$	722	1,323
	Seneca$_{1\text{-}CFA}$	1,590	2,841
Ser5	OPAL$_{RTA}$	6,461	44,773
	Salsa$_{0\text{-}1\text{-}CFA}$	722	1,323
	Salsa$_{1\text{-}CFA}$	1,590	2,841
	Seneca$_{0\text{-}1\text{-}CFA}$	722	1,323
	Seneca$_{1\text{-}CFA}$	1,590	2,841
Ser6	Salsa$_{0\text{-}1\text{-}CFA}$	546	940
	Salsa$_{1\text{-}CFA}$	1,068	1,718
	Seneca$_{0\text{-}1\text{-}CFA}$	546	940
	Seneca$_{1\text{-}CFA}$	1,068	1,718
Ser7	OPAL$_{RTA}$	6,458	44,763
	Salsa$_{0\text{-}1\text{-}CFA}$	722	1,323
	Salsa$_{1\text{-}CFA}$	1,590	2,841
Ser7	Seneca$_{0\text{-}1\text{-}CFA}$	722	1,323
	Seneca$_{1\text{-}CFA}$	1,590	2,841
Ser8	Salsa$_{0\text{-}1\text{-}CFA}$	729	1,333
	Salsa$_{1\text{-}CFA}$	1,601	2,855
Ser8	Seneca$_{0\text{-}1\text{-}CFA}$	729	1,333
	Seneca$_{1\text{-}CFA}$	1,601	2,855
	Soot$_{CHA}$	17,570	261,274
	Soot$_{RTA}$	17,449	259,257
Ser9	OPAL$_{RTA}$	6,463	44,775
	Salsa$_{0\text{-}1\text{-}CFA}$	724	1,325
	Salsa$_{1\text{-}CFA}$	1,592	2,843
	Seneca$_{0\text{-}1\text{-}CFA}$	724	1,325
	Seneca$_{1\text{-}CFA}$	1,592	2,843
	Soot$_{CHA}$	17,570	261,302
	Soot$_{RTA}$	17,449	259,286

spurious nodes and edges. A more recent work, Salsa, also produced call graphs with reasonable sizes, very similar to ours. However, this is because the test cases in the CATS dataset are rather simple; they are up to two classes that exercise one custom call back method at a time. As we will discuss in the next subsection, Salsa's ability to create sound call graphs is greatly diminished when using the source code of real software projects.

5.1.2 Dataset #2: XCorpus Dataset Figure 7 depicts the percentage of edges in the runtime call graph of the projects, that are *missing* on the static call graph computed by each approach. From this chart, we notice that Seneca ***outperformed*** Wala and Salsa. Our approach has ***less*** missing edges compared to other the approaches, *i.e.*, it is able to soundly infer hidden paths through serialization callbacks.

For the *castor* project, Seneca ***did not miss any runtime edge***. In contrast, Wala and Salsa (`0-1-CFA` and `1-CFA`) missed 4.3% of the runtime edges. Seneca$_{0\text{-}1\text{-}CFA}$ also did not miss any runtime edges for two other projects (*james*, and *jpf*), whereas Wala$_{0\text{-}1\text{-}CFA}$ and Salsa$_{0\text{-}1\text{-}CFA}$ missed 8.7% and 5.4% of edges, respectively. The biggest improvements in comparison to other approaches were observed for the test cases created for the *jgraph*, *openjms*, *log4j*, and *xalan* projects. The percentage difference between Seneca and Wala as well as Salsa ranged from 5% to 23.4%.

When inspecting the edges that Seneca missed, we observed that these edges were *unrelated* to serialization callbacks. That is, these were edges to which the underlying pointer analysis algorithm cannot soundly infer the points-to sets of variables. For example, we observed edges that were missed because instructions were using reflection to invoke methods. These were constructs that the underlying `0-1-CFA` and `1-CFA` pointer analysis provided by Wala (our baseline framework) could not correctly infer the dispatch.

One of the reasons as to why Salsa performed similar to Seneca with the CATS test suite but performed poorly on the XCorpus dataset has to do with its inability to compute potential method dispatches from classes in the classpath. As described in their work [Santos et al. 2021, 2020], the approach relies on downcasts of objects to infer what are the object(s) being deserialized. When downcasts are unavailable, the approach relies on a simple approach of computing all possible

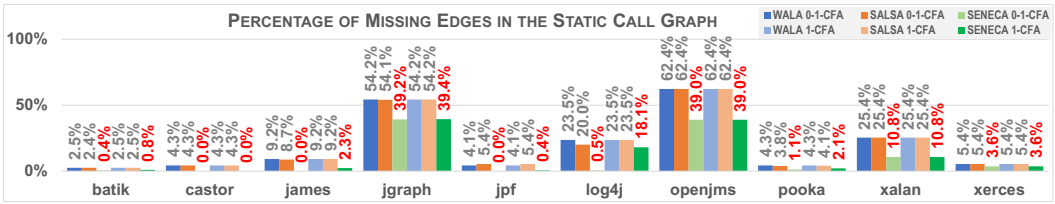

Fig. 7. Percentage of missing edges in the static call graphs computed by WALA, SALSA, and SENECA

dispatches, but limited to classes on the application scope. Our approach, on the other hand, follows Java's serialization specification and includes *all* classes in the classpath, irrespective of its scope (*i.e.*, extension, primordial, or application scope).

Summary of Findings for RQ1

- Our experiments showed that our approach **improved** a call graphs' soundness with respect to serialization-related features. It added nodes and edges in the call graph that could arise at runtime during serialization and deserialization of objects.
- Our approach **passed all test cases**, whereas other approaches, namely Soot_{RTA}, Soot_{RTA} passed only 2, and OPAL_{RTA} passed 5.
- The only call graph construction algorithms used by Soot, and Opal that provided **partial** support for serialization used algorithms that only rely on the method's signatures for dispatch (*i.e.*, CHA and RTA). Hence, they created much larger call graphs because they introduced spurious nodes and edges.
- Although SALSA, a recently published work, also passed all the test cases in the CATS test suite, it failed to soudnly infer the callbacks in real applications from the XCorpus dataset.

5.2 RQ2: Precision

This section describes the evaluation results of the precision of the call graphs computed by SENECA.

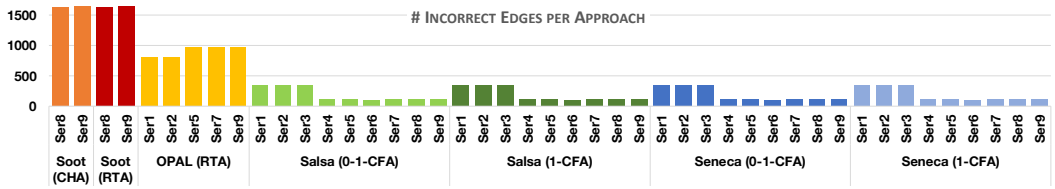

Fig. 8. Number of incorrect edges for the test cases from the CATS test suite.

5.2.1 Dataset #1: CATS Figure 8 depicts the number of edges in the *static* call graph that were not present in the *runtime* call graph for the test cases in the CATS test suite [Reif et al. 2019]. As shown in this chart, SENECA was able to provide full support for serialization callbacks (passing all test cases, see Table 4) while maintaining reasonably sized call graphs. Compared to Soot and OPAL, the derived call graphs were far more *imprecise*. While OPAL and SOOT had over 800 imprecise edges, SENECA had between 95 and 343 incorrect edges.

This comparison also shows that SALSA's performance was similar to SENECA. As explained in the previous section, however, this similar performance is caused by the fact that the programs in the

CATS test suite are small, which does not include scenarios where Salsa's unsound assumptions fall short.

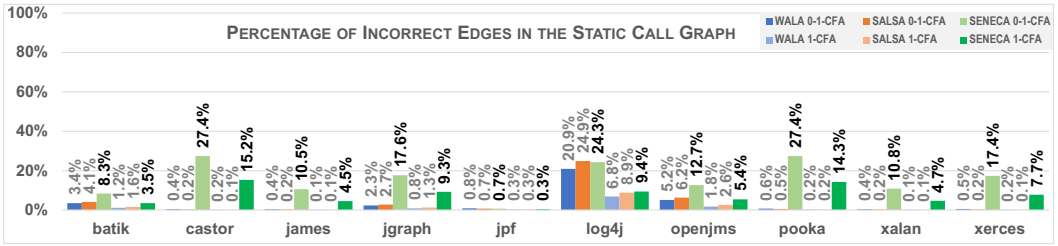

Fig. 9. Percentage of incorrect edges (*i.e.*, edges in the runtime CG not in the static CG) for each approach

5.2.2 Dataset #2: XCorpus Dataset Figure 9 plots the percentage of edges that are in the runtime call graph, but that are not in the static call graph of each approach. As observed on this chart, unsurprisingly, increasing the soundness of the call graph also increased the number of imprecise edges (*i.e.*, edges that did not arise at runtime). The increase of missed edges is comparable to the one by Salsa.

When we inspected the imprecise edges, we noticed that those were related to serialization nodes, *i.e.*, cases in which our call graph included *all* possible objects that can be serialized. Indeed, as our test cases serialized only *one* object at a time, all these edges are deemed as incorrect. However, as the Java API allows the deserialization of arbitrary types (*i.e.*, any serializable type available on the class path), the edges in Seneca could arise at runtime if an object being read uses one of the other serializable classes (other than the one from the test case).

Summary of Findings for RQ2

– Our experiments showed that an increase in soundness included edges that are not in the runtime call graph. However, although these edges did not arise at runtime, they still could be possible (feasible) chains of method invocation since Java's serialization API allows any serializable class in the classpath to be read from an input stream.

5.3 RQ3: Performance

We measured the running time observed when computing the call graphs using Wala, Salsa, and Seneca, configured with `0-1-CFA` and `1-CFA` pointer analysis policies. The results for these experiments are shown in Figure 10. As we would expect, Seneca takes longer to compute call graphs as it has to process nodes and edges related serialization.

The observed differences, however, do not hinder the overall scalability of the approach. The approach still finishes within seconds of execution. Moreover, when further inspecting the worklist of our algorithm, we noticed that Seneca incurs between 3–6 extra iterations over Wala's worklist. These extra iterations along with the taint analysis are the root cause for the extra running time needed for Seneca to finish.

Summary of Findings for RQ3

– Seneca's performance, when evaluated against an established benchmark, has been found to not induce great overhead on the underlying call graph construction approach. This makes Seneca a viable option for developers and researchers in need of sound call graph for analyzing programs that heavily use serialization constructs.

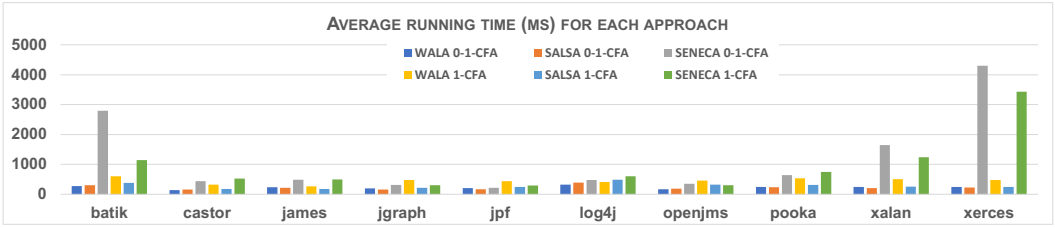

Fig. 10. The total running time (milliseconds) that it took each approach to compute a call graph.

5.4 RQ4: Usefulness for Vulnerability Detection

We have implemented a client analyses that attempts to find vulnerable paths caused by untrusted object deserialization in a program. We then verified how well this client analysis could detect vulnerable paths by comparing its performance using the call graph computed by SALSA and the one generated by SENECA. The results for this experiment are shown in Table 6.

Table 6. Number vulnerable paths found by a client analyses that used SALSA's and SENECA's call graphs

	FileUpload				Vaadin				Wicket			
	SALSA		SENECA		SALSA		SENECA		SALSA		SENECA	
	0-1-CFA	1-CFA	0-1-CFA	1-CFA	0-1-CFA	1-CFA	0-1-CFA	1-CFA	0-1-CFA	1-CFA	0-1-CFA	1-CFA
# Vuln. Paths	0	0	14	12	0	0	4	0	0	0	20	34

As shown in this table, SALSA's call graphs were not suitable for performing vulnerability detection. The key issue lies on the unsoundness of SALSA. This approach relies on type casts (downcasts) to infer what object is being deserialized from a stream. However, as explained in Section 2.2.1, untrusted object deserialization vulnerabilities are caused by the ability of an attacker to craft arbitrary objects using *any* serializable class available in the classpath. Thus, even if the program performs a downcast over the serialized object, the exploit would have been executed anyway, as the vulnerability arises *during* deserialization and not after it.

Unlike SALSA, our approach was able to find vulnerable paths within our allocated time budget (of 15 minutes and up to 15 call graph nodes in a path). The identified paths included the vulnerable paths from previously disclosed gadget chains, documented on the YSoSerial repository of deserialization exploits [Frohoff 2018].

Summary of Findings for RQ4

- We showed the benefits of a sound call graph with respect to deserialization by implement a client static analysis that detect vulnerable paths caused by unstruted object deserialization. Our results showed that while SENECA is able to find previously disclosed vulnerable paths, an existing approach (SALSA) falls short in generating call grapsh that can infer these hidden vulnerable paths.
- The experiments highlight the importance of building call graphs that are sound with respect to deserialization features and demonstrate that SENECA can be suitable for downstream analyses that requires the handling of serialization constructs in a sound fashion.

6 Related Work

This section discusses relevant works related to object deserialization and call graph construction.

6.1 Call graph construction & Taming Challenging Programming Features

Call graphs are a core data structure for multiple analyses. Thus, previous works focused on devising algorithms for their construction. Among these works, we have CHA [Dean et al. 1995] and RTA [Bacon and Sweeney 1996], which are two well-known algorithms that over-approximates possible call paths by relying on methods' signatures. Since these algorithms are overly conservative, multiple works discussed frameworks to make them more precise [Grove and Chambers 2001; Grove et al. 1997; Tip and Palsberg 2000]. Moreover, previous research also focused on creating application-only call graphs, that disregard unnecessary library classes, while keeping on the graph the nodes and edges that are important for the underlying analysis [Ali and Lhoták 2012]. In this paper, we focused on solving the challenge of computing call graphs that are sound concerning object serialization and deserialization.

Previous research on static analysis also explored the challenges involving supporting reflection features [Bodden et al. 2011; Li et al. 2014, 2019; Smaragdakis et al. 2015], dynamic proxies [Fourtounis et al. 2018], enterprise frameworks [Antoniadis et al. 2020] and RMI-based programs [Sharp and Rountev 2006]. These approaches involve making assumptions when performing the analysis, to create analyses that are not overly imprecise. Unlike these prior works, however, we focused on object deserialization that has its own unique challenges, as described in Section 2.3.

6.2 Empirical Studies on Call graphs

Multiple characteristics of call graphs (*e.g.*, precision, soundness, performance, and recall) have been widely studied in the past [Ali et al. 2019; Kummita et al. 2021; Murphy et al. 1998; Sui et al. 2020]. Murphy *et al.* [Murphy et al. 1998] studied multiple call graph construction approaches for C programs, finding discrepancies among the generated call graphs across different approaches. Sui *et al.* [Sui et al. 2018] focused on the support for dynamic language features, aiming to create a benchmark for dynamic features for Java.

There is a line of research that explored call graph's soundness of Java (or JVM-like) programs [Ali et al. 2019; Reif et al. 2019, 2018]. In particular, recent empirical studies [Reif et al. 2019, 2018] show that although serialization-related features are widely used, they are not well-supported in existing approaches. Thus, we built an approach to enhance existing points-to analysis to support the construction of sound call graphs with respect to serialization-related features.

6.3 Pointer Analysis

Many works explored the problem of performing pointer analysis of programs [Bastani et al. 2019; Feng et al. 2015; Heintze and Tardieu 2001; Hind 2001; Kastrinis and Smaragdakis 2013; Lhoták and Hendren 2006; Rountev et al. 2001; Smaragdakis and Kastrinis 2018]. These approaches focus on computing over- or under-approximations to improve one or more aspects of the analysis, such as its soundness, precision, performance, and scalability. Existing pointer analysis approaches make the sets finite such that the problem can be algorithmically solvable. In this paper, however, we focus on aiding points to analysis to soundly handle serialization-related features in a program, which are currently not well-supported because it relies on reflection [Reif et al. 2018].

6.4 Detecting Untrusted Object Deserialization

More recently there were approaches published that aimed at detecting untrusted object deserialization for PHP [Koutroumpouchos et al. 2019; Shahriar and Haddad 2016] and .NET [Shcherbakov and Balliu 2021]. Shcherbakov and Balliu [Shcherbakov and Balliu 2021] described an approach to semi-automatically detect and exploit object injection vulnerabilities .NET applications. It relies on existing publicly available gadgets to perform the detection and exploitation. Koutroumpouchos *et*

al. described ObjectMap [Koutroumpouchos et al. 2019] which is tool that performs black-box analysis of Web applications to pinpoint potential insecure deserialization vulnerabilities. It works by inserting payloads into the parameters of HTTP GET/POST requests and then monitoring the target web application for errors to infer whether the application is vulnerable or not.

Recent works [Cao et al. 2023; Haken 2018; Rasheed and Dietrich 2020] focused on deserialization vulnerabilities in Java programs. Rasheed and Dietrich [Rasheed and Dietrich 2020] described a hybrid approach that first performs a static analysis of a Java program to find potential call chains that can lead to sinks, where reflective method calls are made. It then uses the results of the static analysis to perform fuzzing in order to generate malicious objects.

Unlike these prior works, we aimed to create an approach that can create sound call graphs with respect to serialization-related features. Our call graph is intended to be used by downstream client analyses, including, but not limited to, vulnerability detection.

6.5 Studies on Serialization and Deserializations

In the past few years, there was a spike of vulnerabilities associated with deserialization of objects [Cifuentes et al. 2015]. Thus, existing works also studied vulnerabilities rooted at untrusted deserialization vulnerabilities [Dietrich et al. 2017a; Peles and Hay 2015]. Pele *et al.* [Peles and Hay 2015] conducted an empirical investigation of deserialization of pointers that lead to vulnerabilities in Android applications and SDKs. Dietrich *et al.* [Dietrich et al. 2017a] demonstrated how seemingly innocuous objects trigger vulnerabilities when deserialized, leading to denial of service attacks. In this paper, we describe an approach that could help client analyses focused on detecting instances of untrusted object deserialization.

7 Conclusion

We presented an approach to support the static analysis of serialization-related features in Java programs. It works under the assumption that only classes in the classpath are serialized/deserialized, all of their instance fields are non-nulls and can be allocated with any type that is safe. By applying these assumptions and relying on API modeling, our approach adds synthetic nodes into a previously computed call graph to improve its soundness with respect to serialization-related features.

We evaluated our approach with respect to its *soundness* (RQ1), *precision* (RQ2), *performance* (RQ3), and *usefulness* for a downstream client analysis (RQ4). We used 9 programs from the CATS Test Suite [Reif et al. 2018] and 10 projects from the XCorpus dataset [Dietrich et al. 2017b]. We compared our approach soundness and precision against off-the-shelf construction algorithms available on Soot [Vallée-Rai et al. 1999], Wala [IBM [n.d.]], OPAL [Eichberg and Hermann 2014] and Doop [Bravenboer and Smaragdakis 2009].

In our experiments, we found that only the call graphs that used CHA or RTA could (partially) infer the callback methods that could arise at runtime. Our approach, on the other hand, provided support for all the callback methods in the serialization and deserialization . In an analysis by comparing runtime call graphs with the statically build call graphs, our approach introduced less spurious edges. Finally, by measuring the running times of our approach, compared with its counterpart call graph construction algorithm (Salsa and Wala), we found that our approach did not incur significant overhead.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. CNS-1816845 and Grant No. CCF-1943300. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

References

2023. TensorFlow. https://www.tensorflow.org [Online; accessed 21. Oct. 2023].

Karim Ali, Xiaoni Lai, Zhaoyi Luo, Ondrej Lhotak, Julian Dolby, and Frank Tip. 2019. A Study of Call Graph Construction for JVM-Hosted Languages. *IEEE Transactions on Software Engineering* (2019), 1–1. https://doi.org/10.1109/TSE.2019.2956925

Karim Ali and Ondřej Lhoták. 2012. Application-only call graph construction. In *European Conference on Object-Oriented Programming*. Springer, 688–712.

Anastasios Antoniadis, Nikos Filippakis, Paddy Krishnan, Raghavendra Ramesh, Nicholas Allen, and Yannis Smaragdakis. 2020. Static analysis of Java enterprise applications: frameworks and caches, the elephants in the room.. In *PLDI*. 794–807.

Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-Sensitive and Lifecycle-Aware Taint Analysis for Android Apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. ACM, New York, NY, USA, 259–269. https://doi.org/10.1145/2594291.2594299

David F Bacon and Peter F Sweeney. 1996. Fast static analysis of C++ virtual function calls. In *Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. 324–341. https://doi.org/10.1145/236337.236371

Osbert Bastani, Rahul Sharma, Lazaro Clapp, Saswat Anand, and Alex Aiken. 2019. Eventually Sound Points-To Analysis with Specifications. In *33rd European Conference on Object-Oriented Programming (ECOOP 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. https://doi.org/10.4230/LIPIcs.ECOOP.2019.11

Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. 2011. Taming Reflection: Aiding Static Analysis in the Presence of Reflection and Custom Class Loaders. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE'11)*. ACM, New York, NY, USA, 241–250. https://doi.org/10.1145/1985793.1985827

Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly Declarative Specification of Sophisticated Points-to Analyses. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications* (Orlando, Florida, USA) *(OOPSLA '09)*. Association for Computing Machinery, New York, NY, USA, 243–262. https://doi.org/10.1145/1640089.1640108

Sicong Cao, Biao He, Xiaobing Sun, Yu Ouyang, Chao Zhang, Xiaoxue Wu, Ting Su, Lili Bo, Bin Li, Chuanlei Ma, et al. 2023. ODDFUZZ: Discovering Java Deserialization Vulnerabilities via Structure-Aware Directed Greybox Fuzzing. *arXiv preprint arXiv:2304.04233* (2023).

Cristina Cifuentes, Andrew Gross, and Nathan Keynes. 2015. Understanding Caller-Sensitive Method Vulnerabilities: A Class of Access Control Vulnerabilities in the Java Platform. In *Proceedings of the 4th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis* (Portland, OR, USA) *(SOAP 2015)*. ACM, New York, NY, USA, 7–12. https://doi.org/10.1145/2771284.2771286

Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Trans. Program. Lang. Syst.* 13, 4 (1991), 451–490. https://doi.org/10.1145/115372.115320

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. https://doi.org/10.1007/3-540-49538-X_5

Jens Dietrich, Kamil Jezek, Shawn Rasheed, Amjed Tahir, and Alex Potanin. 2017a. Evil Pickles: DoS Attacks Based on Object-Graph Engineering. In *31st European Conference on Object-Oriented Programming (ECOOP 2017)*, Vol. 74. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 10:1–10:32. https://doi.org/10.4230/LIPIcs.ECOOP.2017.10

Jens Dietrich, Henrik Schole, Li Sui, and Ewan Tempero. 2017b. XCorpus – An executable Corpus of Java Programs. *Journal of Object Technology* 16, 4 (Aug. 2017), 1:1–24. https://doi.org/10.5381/jot.2017.16.4.a1

Julian Dolby, Mandana Vaziri, and Frank Tip. 2007. Finding bugs efficiently with a SAT solver. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. 195–204.

Michael Eichberg. 2020. JCG - SerializableClasses. https://bitbucket.org/delors/jcg/src/master/jcg_testcases/src/main/resources/Serialization.md. (Accessed on 06/01/2020).

Michael Eichberg and Ben Hermann. 2014. A Software Product Line for Static Analyses: The OPAL Framework. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis* (Edinburgh, United Kingdom) *(SOAP '14)*. Association for Computing Machinery, New York, NY, USA, 1–6. https://doi.org/10.1145/2614628.2614630

William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. 2014. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. *ACM Trans. Comput. Syst.* 32, 2, Article 5 (June 2014), 29 pages. https://doi.org/10.1145/2619091

A. Feldthaus, M. Schäfer, M. Sridharan, J. Dolby, and F. Tip. 2013. Efficient construction of approximate call graphs for JavaScript IDE services. In *2013 35th International Conference on Software Engineering (ICSE)*. 752–761. https://doi.org/10.1109/ICSE.2013.6606621

Yu Feng, Xinyu Wang, Isil Dillig, and Thomas Dillig. 2015. Bottom-Up Context-Sensitive Pointer Analysis for Java. In *Programming Languages and Systems - 13th Asian Symposium, APLAS 2015, Pohang, South Korea, November 30 - December 2, 2015, Proceedings (Lecture Notes in Computer Science, Vol. 9458)*, Xinyu Feng and Sungwoo Park (Eds.). Springer, 465–484. https://doi.org/10.1007/978-3-319-26529-2_25

George Fourtounis, George Kastrinis, and Yannis Smaragdakis. 2018. Static analysis of Java dynamic proxies. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 209–220.

Chris Frohoff. 2018. frohoff/ysoserial: A proof-of-concept tool for generating payloads that exploit unsafe Java object deserialization. https://github.com/frohoff/ysoserial. (Accessed on 05/26/2018).

David Grove and Craig Chambers. 2001. A framework for call graph construction algorithms. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 23, 6 (2001), 685–746.

David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers. 1997. Call graph construction in object-oriented languages. In *Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA'97)*. ACM, New York, NY, USA, 108–124. https://doi.org/10.1145/263698.264352

Ian Haken. 2018. Automated Discovery of Deserialization Gadget Chains.

Nevin Heintze and Olivier Tardieu. 2001. Demand-driven pointer analysis. *ACM SIGPLAN Notices* 36, 5 (2001), 24–34. https://doi.org/10.1145/381694.378802

Michael Hind. 2001. Pointer analysis: Haven't we solved this problem yet?. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. 54–61. https://doi.org/10.1145/379605.379665

Stephen Hines, Prasad Kulkarni, David Whalley, and Jack Davidson. 2005. Using De-Optimization to Re-Optimize Code *(EMSOFT '05)*. Association for Computing Machinery, New York, NY, USA, 114–123. https://doi.org/10.1145/1086228.1086251

IBM. [n.d.]. T.J. Watson Libraries for Analysis (WALA). http://wala.sourceforge.net/wiki/index.php/Main_Page. (Accessed on 06/05/2020).

N. Jovanovic, C. Kruegel, and E. Kirda. 2006. Pixy: a static analysis tool for detecting Web application vulnerabilities. In *2006 IEEE Symposium on Security and Privacy (S P'06)*. 6 pp. – 263. https://doi.org/10.1109/SP.2006.29

George Kastrinis and Yannis Smaragdakis. 2013. Hybrid context-sensitivity for points-to analysis. *ACM SIGPLAN Notices* 48, 6 (2013), 423–434. https://doi.org/10.1145/2499370.2462191

R. Khatchadourian, Y. Tang, M. Bagherzadeh, and S. Ahmed. 2019. Safe Automated Refactoring for Intelligent Parallelization of Java 8 Streams. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 619–630. https://doi.org/10.1109/ICSE.2019.00072

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.

Sriteja Kummita, Goran Piskachev, Johannes Späth, and Eric Bodden. 2021. Qualitative and Quantitative Analysis of Callgraph Algorithms for Python. In *2021 International Conference on Code Quality (ICCQ)*. IEEE, 1–15. https://doi.org/10.1109/ICCQ51190.2021.9392986

Davy Landman, Alexander Serebrenik, and Jurgen J. Vinju. 2017. Challenges for Static Analysis of Java Reflection: Literature Review and Empirical Study. In *Proceedings of the 39th International Conference on Software Engineering* (Buenos Aires, Argentina) *(ICSE'17)*. IEEE, New York, NY, USA, 507–518. https://doi.org/10.1109/ICSE.2017.53

Ondřej Lhoták and Laurie Hendren. 2006. Context-sensitive points-to analysis: is it worth it?. In *International Conference on Compiler Construction*. Springer, 47–64. https://doi.org/10.1007/11688839_5

Yue Li, Tian Tan, Yulei Sui, and Jingling Xue. 2014. Self-Inferencing Reflection Resolution for Java. In *Proceedings of the 28th European Conference on ECOOP 2014 — Object-Oriented Programming - Volume 8586*. Springer-Verlag, Berlin, Heidelberg, 27–53. https://doi.org/10.1007/978-3-662-44202-9_2

Yue Li, Tian Tan, and Jingling Xue. 2019. Understanding and analyzing Java reflection. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 28, 2 (2019), 1–50. https://doi.org/10.1145/3295739

Liu Ping, Su Jin, and Yang Xinfeng. 2011. Research on software security vulnerability detection technology. In *Proceedings of 2011 International Conference on Computer Science and Network Technology*, Vol. 3. 1873–1876. https://doi.org/10.1109/ICCSNT.2011.6182335

Alvaro Muñoz and Christian Schneider. 2018. Serial killer: Silently pwning your Java endpoints. http://www.slideshare.net/cschneider4711/owasp-benelux-day-2016-serial-killer-silently-pwning-your-java-endpoints. (Accessed on 11/15/2019).

Gail C Murphy, David Notkin, William G Griswold, and Erica S Lan. 1998. An empirical study of static call graph extractors. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 7, 2 (1998), 158–191.

Oracle. 2010. Java Object Serialization Specification - version 6.0. https://docs.oracle.com/javase/8/docs/platform/serialization/spec/serialTOC.html. (Accessed on 04/07/2020).

Or Peles and Roee Hay. 2015. One Class to Rule Them All: 0-Day Deserialization Vulnerabilities in Android. In *9th USENIX Workshop on Offensive Technologies (WOOT 15)*. USENIX Association, Washington, D.C., 12.

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*. 1209–1213.

Michael Reif. 2023. mreif/jcg - Docker Image | Docker Hub. https://hub.docker.com/r/mreif/jcg [Online; accessed 20. Oct. 2023].

Michael Reif, Florian Kübler, Michael Eichberg, Dominik Helm, and Mira Mezini. 2019. Judge: Identifying, Understanding, and Evaluating Sources of Unsoundness in Call Graphs. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. ACM, New York, NY, USA, 251–261. https://doi.org/10.1145/3293882.3330555

Michael Reif, Florian Kübler, Michael Eichberg, and Mira Mezini. 2018. Systematic Evaluation of the Unsoundness of Call Graph Construction Algorithms for Java. In *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops (ISSTA'18)*. ACM, New York, NY, USA, 107–112. https://doi.org/10.1145/3236454.3236503

Henry Gordon Rice. 1953. Classes of recursively enumerable sets and their decision problems. *Trans. Amer. Math. Soc.* 74, 2 (1953), 358–366.

Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. 1988. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 12–27.

Atanas Rountev, Ana Milanova, and Barbara G Ryder. 2001. Points-to analysis for Java using annotated constraints. *ACM SIGPLAN Notices* 36, 11 (2001), 43–55. https://doi.org/10.1145/504311.504286

Joanna C. S. Santos, Reese A. Jones, Chinomso Ashiogwu, and Mehdi Mirakhorli. 2021. Serialization-Aware Call Graph Construction. In *Proceedings of the 10th ACM SIGPLAN International Workshop on the State of the Art in Program Analysis*.

Joanna C. S. Santos, Reese A. Jones, and Mehdi Mirakhorli. 2020. Salsa: Static Analysis of Serialization Features. In *Proceedings of the 22th ACM SIGPLAN International Workshop on Formal Techniques for Java-Like Programs (FTfJP '20)* (Virtual) *(FTfJP 2020)*. ACM, New York, NY, USA, 18–25. https://doi.org/10.1145/3427761.3428343

Imen Sayar, Alexandre Bartel, Eric Bodden, and Yves Le Traon. 2023. An in-depth study of java deserialization remote-code execution exploits and vulnerabilities. *ACM Transactions on Software Engineering and Methodology* 32, 1 (2023), 1–45.

Christian Schneider and Alvaro Muñoz. 2016. Java Deserialization Attacks. https://owasp.org/www-pdf-archive/GOD16-Deserialization.pdf. (Accessed on 11/15/2019).

Edward J Schwartz, Thanassis Avgerinos, and David Brumley. 2010. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *2010 IEEE symposium on Security and privacy*. IEEE, 317–331.

Hossain Shahriar and Hisham Haddad. 2016. Object injection vulnerability discovery based on latent semantic indexing. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*. 801–807.

M. Sharp and A. Rountev. 2006. Static Analysis of Object References in RMI-Based Java Software. *IEEE Transactions on Software Engineering* 32, 9 (2006), 664–681. https://doi.org/10.1109/TSE.2006.93

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*.

Yannis Smaragdakis, George Balatsouras, George Kastrinis, and Martin Bravenboer. 2015. More Sound Static Handling of Java Reflection. In *Programming Languages and Systems*, Xinyu Feng and Sungwoo Park (Eds.). Springer International Publishing, Cham, 485–503. https://doi.org/10.1007/978-3-319-26529-2_26

Yannis Smaragdakis and George Kastrinis. 2018. Defensive Points-To Analysis: Effective Soundness via Laziness. In *32nd European Conference on Object-Oriented Programming (ECOOP 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. https://doi.org/10.4230/LIPIcs.ECOOP.2018.23

NSA Center for Assured Software. 2023. Juliet Java 1.3. https://samate.nist.gov/SARD/test-suites/111 [Online; accessed 1. May. 2022].

Manu Sridharan, Shay Artzi, Marco Pistoia, Salvatore Guarnieri, Omer Tripp, and Ryan Berg. 2011. F4F: Taint Analysis of Framework-Based Web Applications. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Portland, Oregon, USA) *(OOPSLA '11)*. ACM, New York, NY, USA, 1053–1068. https://doi.org/10.1145/2048066.2048145

Manu Sridharan, Satish Chandra, Julian Dolby, Stephen J. Fink, and Eran Yahav. 2013. *Alias Analysis for Object-Oriented Programs*. Springer Berlin Heidelberg, Berlin, Heidelberg, 196–232. https://doi.org/10.1007/978-3-642-36946-9_8

Li Sui, Jens Dietrich, Michael Emery, Shawn Rasheed, and Amjed Tahir. 2018. On the soundness of call graph construction in the presence of dynamic language features-a benchmark and tool evaluation. In *Asian Symposium on Programming Languages and Systems*. Springer, 69–88.

Li Sui, Jens Dietrich, Amjed Tahir, and George Fourtounis. 2020. On the Recall of Static Call Graph Construction in Practice. https://doi.org/10.1145/3377811.3380441

H. Thaller, L. Linsbauer, A. Egyed, and S. Fischer. 2020. Towards Fault Localization via Probabilistic Software Modeling. In *2020 IEEE Workshop on Validation, Analysis and Evolution of Software Tests (VST)*. 24–27. https://doi.org/10.1109/VST50071.2020.9051635

Julian Thomé, Lwin Khin Shar, Domenico Bianculli, and Lionel C Briand. 2017. Joanaudit: A tool for auditing common injection vulnerabilities. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 1004–1008.

Frank Tip and Jens Palsberg. 2000. Scalable propagation-based call graph construction algorithms. In *Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. 281–293.

Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - a Java Bytecode Optimization Framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research* (Mississauga, Ontario, Canada) *(CASCON '99)*. IBM Press, 13.

Marvin Wyrich and Justus Bogner. 2019. Towards an Autonomous Bot for Automatic Source Code Refactoring. In *Proceedings of the 1st International Workshop on Bots in Software Engineering* (Montreal, Quebec, Canada) *(BotSE '19)*. IEEE Press, 24–28. https://doi.org/10.1109/BotSE.2019.00015