Serialization-Aware Call Graph Construction

Joanna C. S. Santos
Rochester Institute of Technology, USA
jds5109@rit.edu

Reese A. Jones
Rochester Institute of Technology, USA
raj8065@rit.edu

Chinomso Ashiogwu
University of Maryland, Baltimore County, USA
cashiog1@umbc.edu

Mehdi Mirakhorli
Rochester Institute of Technology, USA
mxmvse@rit.edu

Abstract

Although call graphs are crucial for inter-procedural analyses, it is challenging to statically compute them for programs with dynamic features. Prior work focused on supporting certain kinds of dynamic features, but serialization-related features are still not very well supported. Therefore, we introduce Salsa, an approach to complement existing points-to analysis with respect to serialization-related features to enhance the call graph's soundness while not greatly affecting its precision. We evaluate Salsa's soundness, precision, and performance using 9 programs from the Java Call graph Assessment & Test Suite (CATS) and 4 programs from the XCorpus dataset. We compared Salsa against off-the-shelf call graph construction algorithms available on Soot, Doop, WALA, and OPAL. Our experiments showed that Salsa improved call graphs' soundness while not greatly affecting their precision. We also observed that Salsa did not incur an extra overhead on the underlying pointer analysis method.

CCS Concepts: • **Software and its engineering** → *Compilers*; **Automated static analysis**; • **Theory of computation** → **Program analysis**.

Keywords: Serialization, Deserialization, Call graphs

ACM Reference Format:
Joanna C. S. Santos, Reese A. Jones, Chinomso Ashiogwu, and Mehdi Mirakhorli. 2021. Serialization-Aware Call Graph Construction. In *Proceedings of the 10th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis (SOAP '21), June 22, 2021, Virtual Event, Canada.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3460946.3464319

1 Introduction

Call graphs [10] are crucial for performing multiple types of inter-procedural analysis, such as vulnerability detection, information flow analysis, and optimization. Building a call graph statically can be challenging when a program uses certain programming language constructs, such as native calls, reflection, and object serialization. These constructs are frequently used to load classes, invoke methods, instantiate objects and extend the programs' functionalities [15, 26]. Ignoring such constructs leads to **unsound** call graphs which miss possible runtime paths [1, 20, 21, 28].

Previous works explored certain features, such as reflection [5, 17, 18, 26], native code [27], and Remote Method Invocation (RMI) [25]. However, as shown by Reif *et al.* [20, 21], there is currently limited support for building call graphs handling *serialization and deserialization of objects*.

Multiple programming languages (*e.g.*, Java, Ruby, Python, PHP, etc) allow objects to be converted into an *abstract representation*, a process called **object serialization**. The reconstruction of an object from its underlying representation is called **object deserialization**. (De)serialization is widely used for inter-process communication [3, 8] and to improve the system's performance, *e.g.*, saving a trained machine learning model to be used later without retraining it.

During object (de)serialization, certain methods from the objects' classes may be invoked, *e.g.*, classes' constructors, getter/setter methods, or methods with specific signatures. These are the **callback methods** of the serialization/deserialization mechanism. State-of-the-art algorithms for Java fall short in having nodes and edges that represent *callback* methods that are invoked during object serialization/deserialization [20, 21]. There are multiple challenges during resolution of these callback method invocations. First, the serialization API relies on "non-trivial" reflective calls that current techniques [15] for taming reflection do not address. Second, the callback methods are invoked on-the-fly as each object and its fields are read from a stream. Thus, the actual invoked callback methods are only known at runtime.

In our prior work [23], we described our early efforts in developing Salsa to provide support for Java serialization features. In this paper, we extend Salsa to create a serialization-aware call graph extractor that provides full support to all the possible runtime paths via call back methods. We also present additional experiments to demonstrate how Salsa improves a call graphs' *soundness* with respect to (de)serialization callbacks without greatly affecting its *precision*. The contributions of this paper are threefold: (i) an improvement to Salsa to provide full support of serialization-related callbacks. (ii) an evaluation of Salsa's *soundness*, *precision*, and *scalability*; (iii) a publicly available implementation on top of WALA[1].

2 Background on Java Serialization API

Java's Serialization API converts an object graph into a *byte stream*. During this process only *data* is serialized (*i.e.*, *non-transient* and *non-static* fields) whereas the code associated with the object's class (*i.e.*, methods) is within the classpath of the receiver [24].

```
 1 class Pet implements Serializable { protected String name; }
 2 class Cat extends Pet{
 3   private void readObject(ObjectInputStream s) { /* ... */ }
 4   private void writeObject(ObjectOutputStream s){ /* ... */ }
 5 }
 6 class Dog extends Pet{
 7   private Object readResolve() { /* ... */ }
 8   private Object writeReplace() { /* ... */ }
 9 }
10 class Shelter implements Serializable{ private List<Pet> pets; }
11 class SerializationExample{
12   public static void main(String[] args) throws Exception {
13     List<Pet> pets = Arrays.asList(new Dog("Max"), new Cat("Joy"));
14     Shelter s1 = new Shelter(pets);
15     FileOutputStream f = new FileOutputStream(new File("pets.txt"));
16     ObjectOutputStream out = new ObjectOutputStream(f);
17     out.writeObject(s1);
18   }
19 }
20 class DeserializationExample{
21   public static void main(String[] args) throws Exception {
22     FileInputStream fs = new FileInputStream(new File("pets.txt"));
23     ObjectInputStream in = new ObjectInputStream(fs);
24     Shelter s2 = (Shelter) in.readObject();
25   }
26 }
```

Listing 1. Object serialization and deserialization example

The classes ObjectInputStream and ObjectOutputStream from the java.io package can be used for deserializing and serializing an object, respectively. The classes can only serialize/deserialize objects whose class implements the Serializable interface. If implemented by Serializable classes, the following methods are invoked by Java during deserialization (1-4) or serialization (5-6): **(1)** void readObjectNoData() (initializes the object's state in the exceptional situation that a receiver has a subclass in its classpath but not its superclass); **(2)** void readObject(ObjectInputStream) (customizes the retrieval of an object's state from the stream); **(3)** Object readResolve() (allows classes to replace a specific instance that is being read from the stream); **(4)** void validateObject() (validates an object after it is deserialized); **(5)** void writeObject(ObjectOutputStream) (customizes the serialization of the object's state); **(6)** Object writeReplace() (replaces the actual object that will be written in the stream);

[1]Salsa is available at: https://github.com/SoftwareDesignLab/Salsa

Demonstrative Example: Listing 1 has four serializable classes[2]: Shelter, Cat, Dog, and Pet (which is the superclass for Dog and Cat classes). Two classes have callback methods (lines 2-5, and 6-9). The code at line 13-17 serializes a Shelter object s1 into a file. 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 in "shelter.txt". Since the object s1 has a list field that contains two objects (a Cat and a Dog instance) the writeObject and writeReplace callbacks are invoked. The main method at line 21 deserializes this object from the file. It creates an ObjectInputStream instance and invokes the method readObject(), which returns an object constructed from the "pets.txt" 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.

3 Salsa Overview

There are two major challenges when constructing a sound call graph with respect to serialization-related features: (i) resolving the invocation of **callback methods** 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 (*e.g.*, if the code snippet in Listing 1 had only the cat instance in the list (line 13), then the calls to readResolve/writeReplace methods in Dog would not be made). To support serialization-related features, Salsa employs an iterative call graph construction framework that involves two major phases: ① A set of iterations over a worklist of methods to create an initial (unsound) call graph using an underlying pointer analysis method; ② An iterative refinement of the initial call graph.

3.1 Phase 1: Initial Call Graph Construction

Salsa takes as input a CSV file with method signatures for **entrypoints**, which are the methods that start the program's execution. Then, it analyzes the program's entrypoints. The result of this step is a set of entrypoint methods m added to the worklist $\mathcal{W}$. The **worklist** $\langle m, c\rangle \in \mathcal{W}$ tracks all the methods m under a context c that have to be traversed and analyzed. A **context** c is an abstraction of a program's state. The entrypoints are assigned a global context $\emptyset$ [28]

Starting from the entrypoint methods, Salsa constructs an initial call graph (*i.e.*, call graph$_0$) using the underlying pointer analysis algorithm selected by the client analysis (*e.g.*, , n-CFA, l-n-CFA, etc). Each method in the worklist is converted into an Intermediary Representation (IR) in Single Static Assignment form (SSA) [6]. Each instruction in this IR is visited following the rules by the underlying pointer analysis algorithm. For a generic formulation for multiple points-to analyses, we point the reader to prior research [28].

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

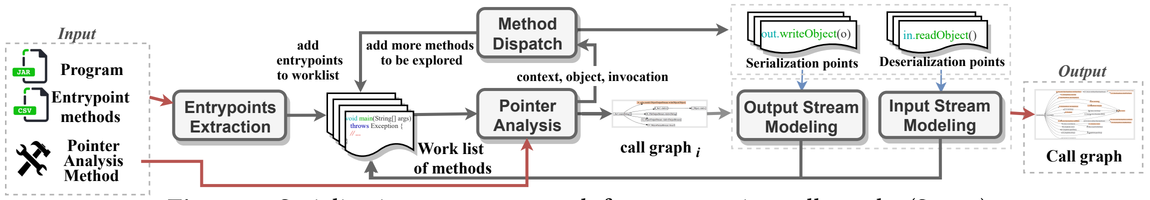

Figure 1. Serialization-aware approach for constructing call graphs (Salsa)

When visiting instance invocations in a method m *i.e.*, x = o.g(a_1,$\cdots$,a_n), Salsa computes the possible dispatches for the method g as follows: targets = $dispatch(pt(\langle o, c\rangle), g)$. The dispatch mechanism takes into account the declared target g and the points-to set for the object o at the current context c. If the declared target is the ObjectOutputStream's writeObject(Object) or the ObjectInputStream's readObject(), then the dispatch function creates a *synthetic method* to model the runtime behavior for the readObject() and writeObject(Object) methods. These callsites are **deserialization** and **serialization points**, respectively.

Salsa creates synthetic methods *without* instructions. Instructions are added during the call graph refinement phase. Calls to synthetic methods are *1-callsite-sensitive* [28] to take into account that the same Object(In|Out)putStream instance can be used to read/write multiple objects. Thus, we want to disambiguate these paths in the call graph. As a result of Phase 1, we obtain the ***initial callgraph*** (g_0) and the ***serialization and deserialization points***.

3.2 Phase 2: Call Graph Refinement

Salsa iteratively refines the current call graph g_i by adding instructions to *synthetic* methods to model the invocation to callbacks. The iterations are done until a fixpoint is reached (*i.e.*, , there are no synthetic methods in need of refinement).

3.2.1 Modeling Object Serialization. For each *instruction* at the serialization points, Salsa obtains the points-to set for the object o passed as the first argument to writeObject (line 2 in Algorithm 1). The points-to set $pt(\langle o, c\rangle)$ indicates the set of allocated types t for the passed object o_1 under context c. For each type $t \in pt(\langle o, c\rangle)$, Salsa adds a type cast instruction to m_s that downcast the writeObject's argument to the type t (line 5). If t implements any of the serialization callbacks (Section 2), Salsa adds an invocation instruction from m_s targeting this *callback* method. Then, Salsa iterates over all non-static fields f from the class t and compute their points-to sets (lines 9-10). If the concrete type(s) allocated to the field contains callback methods, it adds 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. Lastly, Salsa re-adds the synthetic method m_s to the worklist.

3.2.2 Modeling Object Deserialization. Since multiple classes in a classpath (*e.g.*, Java's Swing classes) can implement the Serializable interface, objects within a source stream can be an instance of any of these classes. Hence, there is a high amount of possible calls that would be erroneously included in the resulting call graph. To overcome this, we make the following assumptions while modeling object deserialization: **(1)** there is no dynamic loading of remote classes (closed-world assumption) [19], **(2)** all non-static fields in serializable classes are not null and can be allocated with any type that is safe, **(3)** all downcasts are safe. Assumption #2 ensures that we can soundly infer possible call targets within callback methods made via inner fields (*e.g.*, , line 24 in Listing 1). Assumption #3 is crucial to reduce the points-to sets for fields within serializable classes.

When modeling deserialization, Salsa first traverses the def-use chains of the caller's IR to find any downcasts for the returned deserialized object (line 4 in Algorithm 2):

$$\mathbf{o_{ret}} = \text{in.readObject();} \quad \cdots \quad x = \text{(T)}\ \mathbf{o_{ret}};$$

For each downcast type t (line 4), Salsa adds an allocation instruction to m_s followed by invocations to callbacks implemented by t (if any exists). Next, it iterates over all instance fields of the type and compute the possible serializable classes that are type-safe for the field (lines 9-10). For each possible safe type, it adds a field allocation. Then, if the possible type has a callback method, it adds to m_s: a cast to the possible type (line 15), and an invocation to the callback (line 16). Finally, the synthetic method is re-added to the worklist $\mathcal{W}$.

Algorithm 1: Object serialization modeling

```
    Input: $I$: serialization points (i.e., ObjectOutputStream.writeObject($o_1$));
           $G$: Project's initial call graph;
    Output: Set of refined synthetic models $M_s$
 1  foreach $instruction \in I$ do
 2      $\langle o, c\rangle \leftarrow$ getPointerForArg(1, $instruction$)
 3      $m_s \leftarrow$ declaredTarget($instruction$)
 4      foreach $t \in pt(\langle o, c\rangle)$ do
 5          addTypeCast($m_s$,$t$)
 6          foreach $callback \in callbacks(t)$ do
 7              addInvoke($m_s$, $callback$)
 8          end
 9          foreach $f \in fields(t)$ do
10              foreach $fieldType \in pt(\langle o.f, c\rangle)$ do
11                  foreach $callback \in callbacks(fieldType)$ do
12                      addGetField($m_s$, $f$)
13                      addTypeCast($m_s$, $fieldType$)
14                      addInvoke($m_s$, $callback$)
15                  end
16              end
17          end
18      end
19      addToWorkList($m_s$, $c$)
20  end
```

– **Handling object array/collection fields**: We make an extra assumption that all array/collection fields contains at least one object of each possible type (according to Java's type safety and accessibility rules). This ensures that we soundly infer possible targets for calls whose receiver object is from an array/collection field. To ensure SALSA keeps its soundness promises, it is not *container-sensitive*, *i.e.*, it does not keep different points-to sets for a[i] and a[j] ($i \neq j$).

Algorithm 2: Object deserialization modeling

```
Input: Set of invocation instructions to ObjectInputStream.readObject: $I$;
       Project's initial call graph: $G$;
       Serializable classes in the classpath: $S$;
Output: Set of refined synthetic models $M_s$
foreach $instruction$ in $I$ do
    $\langle o_{ret}, c \rangle \leftarrow$ getPointerForReturnValue($instruction$)
    $m_s \leftarrow$ declaredTarget($instruction$)
    foreach $t \in downcasts(o_{ret})$ do
        $o_i \leftarrow$ addAllocation($m_s$, $t$)
        foreach $callback \in$ callbacks($t$) do
            addInvoke($m_s$, $callback$)
        end
        foreach $f \in fields(t)$ do
            foreach $type \in possibletypes(f)$ do
                addAllocation($m_s$,$o_i.f$, $type$)
                foreach $callback \in$ callbacks($type$) do
                    addGetField($m_s$,$o_i.f$)
                    addTypeCast($m_s$,$o_i.f$,$type$)
                    addInvoke($m_s$,$type.readObject$)
                end
            end
        end
    end
    addToWorkList($m_s$,c)
end
```

4 Evaluation

We focus on the following research questions:

RQ1 *Does SALSA improve a call graph's soundness with respect to serialization features?*

RQ2 *Does SALSA introduce spurious nodes/edges?*

RQ3 *Does SALSA scale to realistic programs?*

We developed SALSA's prototype in Java using WALA. It supports two kinds of pointer analyses: 0-n-CFA or n-CFA (where n can be specified).

4.1 Answering RQ1: Soundness

We use the Java Call-graph Assessment & Test Suite (CATS)[3] to answer RQ1. This test suite was released as part of a recent work [20] that used off-the-shelf call graph construction algorithms available on Soot, Doop, WALA, and OPAL to compare the soundness of the computed call graphs with respect to particular programming language constructs. CATS test suite 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. We run SALSA using three configurations: 0-1-CFA, 1-CFA, and 2-CFA. We compare SALSA against the same algorithms used in the empirical study by Reif *et al.* [20]: SOOT (CHA, RTA, VTA, and Spark), WALA (RTA, 0-CFA, 1-CFA, and 0-1-CFA), DOOP (context-insensitive), and OPAL (RTA).

—**RQ1 Results**: Table 1 reports the programs in which each approach soundly inferred the call graph (✓) and the ones it failed to do so (✗). While SALSA passed ***all*** of the nine test cases, only three other approaches partially provided support for callback methods, *i.e.*, Soot$_{RTA}$ and Soot$_{CHA}$ (2 out of 9) and OPAL$_{RTA}$ (5 out of 9). These algorithms that provided partial support use ***imprecise*** call graph construction algorithms (CHA or RTA). The remaining 7 algorithms did not provide support at all for callback methods. Table 2 compares the call graphs' sizes in terms of nodes and edges. While SALSA constructed call graphs with a number of nodes ranging from 549 to 15,319 and number of edges ranging from 944 to 99,861, the other algorithms ranged from 6650 to 7208 (Opal), from 20027 to 20168 (Soot) in terms of nodes and from 59,039 to 66,175 (Opal) and 327,530 to 329,815 (Soot) in terms of edges. Since Soot$_{RTA}$, Soot$_{CHA}$, and OPAL$_{RTA}$ rely on static types when computing the possible targets of a method invocation, they introduce spurious nodes/edges, thereby increasing the call graph's size.

Table 1. Results from running the test cases from JCG

Approach	Ser1	Ser2	Ser3	Ser4	Ser5	Ser6	Ser7	Ser8	Ser9
SALSA$_{0\text{-}1\text{-CFA}}$	✓	✓	✓	✓	✓	✓	✓	✓	✓
SALSA$_{1\text{-CFA}}$	✓	✓	✓	✓	✓	✓	✓	✓	✓
SALSA$_{2\text{-CFA}}$	✓	✓	✓	✓	✓	✓	✓	✓	✓
OPAL$_{RTA}$	✓	✓	✗	✗	✓	✗	✓	✗	✓
SOOT$_{CHA}$	✗	✗	✗	✗	✗	✗	✗	✓	✓
SOOT$_{RTA}$	✗	✗	✗	✗	✗	✗	✗	✓	✓
SOOT$_{VTA,\ Spark}$ WALA$_{RTA,\ 0\text{-CFA},\ 1\text{-CFA},\ 0\text{-}1\text{-CFA}}$ DOOP$_{CI}$	✗	✗	✗	✗	✗	✗	✗	✗	✗

4.2 Answering RQ2: Precision

A call graph's precision is measured by the number of extra nodes/edges that it contains that do not arise at runtime. We use the *java-callgraph* tool[4] to construct the dynamic call graph for each program in the CATS test suite. We configured this tool with "incl=ser.*,java.io.*" to only track methods whose declaring classes match the regular expressions above. After computing the dynamic call graph, we calculate the number of nodes and edges that appeared in our static call graph but did not appear on the dynamic call graph. We disregard nodes from the static call graph that are not being tracked by the instrumenter (*i.e.*, , that do not match the configuration above). We compare SALSA against Soot$_{RTA,CHA}$, and OPAL$_{RTA}$ because they were the only approaches that provided some support for callback methods.

—**RQ2: Precision Results**: Figure 2 shows the total number of incorrect (spurious) edges in each approach. SALSA exhibited the least amount of incorrect edges. SALSA had 358 incorrect edges on average whereas OPAL, and SOOT had

[3] https://bitbucket.org/delors/cats/src/master/jcg_testcases

[4] https://github.com/gousiosg/java-callgraph

Table 2. Number of nodes and edges in each computed static call graph

Test Case	Ser1				Ser2				Ser3			Ser4			Ser5			
Approach	**Salsa 0-1-CFA**	**Salsa 1-CFA**	**Salsa 2-CFA**	**OPAL RTA**	**Salsa 0-1-CFA**	**Salsa 1-CFA**	**Salsa 2-CFA**	**OPAL RTA**	**Salsa 0-1-CFA**	**Salsa 1-CFA**	**Salsa 2-CFA**	**Salsa 0-1-CFA**	**Salsa 1-CFA**	**Salsa 2-CFA**	**Salsa 0-1-CFA**	**Salsa 1-CFA**	**Salsa 2-CFA**	**OPAL RTA**
# Nodes	799	1932	3641	6650	800	1934	3643	6651	800	1933	3642	1511	5038	15307	1511	5038	15307	7206
# Edges	1629	3640	7100	59039	1631	3642	7102	59042	1630	3641	7101	3563	15321	99847	3563	15321	99847	66173

Test Case	Ser6			Ser7				Ser8					Ser9					
Approach	**Salsa 0-1-CFA**	**Salsa 1-CFA**	**Salsa 2-CFA**	**Salsa 0-1-CFA**	**Salsa 1-CFA**	**Salsa 2-CFA**	**OPAL RTA**	**Salsa 0-1-CFA**	**Salsa 1-CFA**	**Salsa 2-CFA**	**Soot CHA**	**Soot RTA**	**Salsa 0-1-CFA**	**Salsa 1-CFA**	**Salsa 2-CFA**	**OPAL RTA**	**Soot CHA**	**Soot RTA**
# Nodes	549	1072	1728	1171	3775	12186	7203	1516	5047	15319	20168	20027	1513	5040	15309	7208	20168	20027
# Edges	944	1727	2878	2556	12185	92347	66162	3571	15333	99861	329788	327530	3565	15323	99849	66175	329815	327563

899 and 1637, respectively. Moreover, Opal and Soot were between 2.5-4.5x times more imprecise than Salsa.

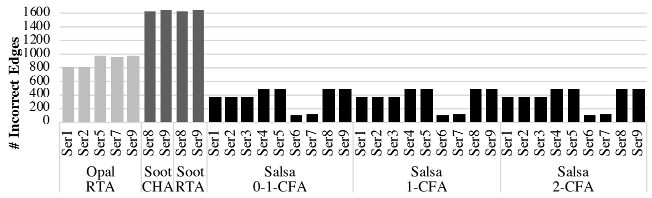

Figure 2. Number of spurious edges in each approach

4.3 Answering RQ3: Performance

To verify the overhead incurred by Salsa, we use 4 programs from the XCorpus dataset [7]. We selected these 4 projects because they match the following criteria: (i) they perform object serialization/deserialization; (ii) there are application serializable classes that provide custom implementation for callback methods. We run the 0-1-CFA and 1-CFA call graph construction algorithms available in WALA ***with*** and ***without*** our serialization-aware approach. For both cases, we configure WALA to consider all declared application methods in the analysis scope as entrypoints. Moreover, we used WALA's list of class exclusions [5]; these classes are ignored during call graph construction. We ran Salsa and WALA on the projects from the XCorpus dataset [7]. We measured the total running time of each approach to compute the call graph. We run these analyses on a machine with a 2.9 GHz Intel Core i7 processor and 16 GB of RAM memory.

Table 3. Performance analysis (running time in seconds)

Project	WALA 0-1-CFA	Salsa 0-1-CFA	WALA 1-CFA	Salsa 1-CFA
log4j-1.2.16	7.44	13.43	23.24	15.25
htmlunit-2.8	5.38	10.37	21.94	16.71
pooka-3.0-080505	29.05	111.53	587.88	156.01
megamek-0.35.18	33.07	66.94	737.05	735.91

—**RQ3: Performance Results**: Table 3 contains these measurements for each project. When using 0-1-CFA, Salsa took longer to execute compared to WALA. This is expected, since Salsa is 1-callsite-sensitive for calls to synthetic methods, as compared to 0-1-CFA. However, when using 1-CFA Salsa took ***less*** time to complete. The reason is that we remove the complexity of analyzing the Object(In|Out)putStream classes, which use multiple other classes to implement the serialization/deserialization protocol. By abstracting these complexities, we reduce the number of analyzed instructions which decreases the number of dataflow constraints that need to be solved by the pointer analysis.

5 Related Work

Multiple works discussed frameworks to construct call graphs and make them more precise [10, 11, 31]. Previous research also focused on creating application-only call graphs, that disregard unnecessary library classes, while keeping nodes and edges that are important for the underlying analysis [2]. We focused on computing call graphs that are sound concerning (de)serialization callbacks.

Multiple call graphs' characteristics (*e.g.*, precision, soundness, performance, etc) have been studied [1, 20, 21, 30]. Sui *et al.* [29] focused on the support for dynamic language features, aiming to create a benchmark for dynamic features for Java. Other works explored call graph's soundness of JVM-like programs [1, 20, 21]. Reif *et al.* [20, 21] showed that although serialization-related features are widely used, they are not well supported in existing approaches.

Many works [4, 9, 12–14, 16, 22, 27] explored the problem of performing pointer analysis whose one main client is call graph extraction. They compute over-/under-approximations to improve one or more aspects, such as its soundness, precision, performance, and scalability. In this paper, however, we focus on aiding points to analysis to soundly handle serialization-related callbacks.

6 Conclusion

We described an approach to support serialization-related features in Java programs. We evaluated Salsa with respect to its *soundness* (RQ1), *precision* (RQ2), and *performance* (RQ3). We found that only the call graphs that used CHA or RTA could (partially) infer the callback methods that could arise at runtime. Salsa, on the other hand, provided support for all the callback methods in the serialization and deserialization, while not greatly affecting its precision and not incurring significant overhead.

Acknowledgments

This work was partially funded by the US National Science Foundation under grants number CNS-1816845 and CCF-1943300.

[5] https://github.com/wala/WALA/wiki/Pointer-Analysis

References

[1] 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

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

[3] 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 (PLDI '14)*. ACM, 259âĂŞ269. https://doi.org/10.1145/2594291.2594299

[4] 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*. https://doi.org/10.4230/LIPIcs.ECOOP.2019.11

[5] 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, 241–250. https://doi.org/10.1145/1985793.1985827

[6] 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

[7] Jens Dietrich, Henrik Schole, Li Sui, and Ewan Tempero. 2017. 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

[8] 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

[9] Yu Feng, Xinyu Wang, Isil Dillig, and Thomas Dillig. 2015. Bottom-Up Context-Sensitive Pointer Analysis for Java. In *13th Asian Symposium Programming Languages and Systems (APLAS)*. Springer, 465–484. https://doi.org/10.1007/978-3-319-26529-2_25

[10] David Grove and Craig Chambers. 2001. A framework for call graph construction algorithms. *ACM Trans. Program. Lang. Syst. (TOPLAS)* 23, 6 (2001), 685–746.

[11] 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, 108–124. https://doi.org/10.1145/263698.264352

[12] 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

[13] 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

[14] 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

[15] Davy Landman, Alexander Serebrenik, and Jurgen J. Vinju. 2017. Challenges for Static Analysis of Java Reflection: Literature Review and Empirical Study. In *Proceedings of the 39th International Conference on Software Engineering (ICSE'17)*. IEEE, 507âĂŞ518. https://doi.org/10.1109/ICSE.2017.53

[16] 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

[17] Yue Li, Tian Tan, Yulei Sui, and Jingling Xue. 2014. Self-Inferencing Reflection Resolution for Java. In *Proceedings of the 28th European Conference on Object-Oriented Programming (ECOOP'14)*. Springer-Verlag, 27âĂŞ53. https://doi.org/10.1007/978-3-662-44202-9_2

[18] 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

[19] Benjamin Livshits, John Whaley, and Monica S. Lam. 2005. Reflection Analysis for Java. In *Proceedings of the Third Asian Conference on Programming Languages and Systems (APLASâĂŽ05)*. Springer-Verlag, Berlin, Heidelberg, 139âĂŞ160. https://doi.org/10.1007/11575467_11

[20] 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 (ISSTA 2019)*. ACM, 251âĂŞ261. https://doi.org/10.1145/3293882.3330555

[21] 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, 107–112. https://doi.org/10.1145/3236454.3236503

[22] 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

[23] 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)*. ACM, 18–25. https://doi.org/10.1145/3427761.3428343

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

[25] 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

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

[27] 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

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

[29] 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.

[30] 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

[31] 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.