SALSA: Static Analysis of Serialization Features

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

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

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

Abstract

Static analysis has the advantage of reasoning over multiple possible paths. Thus, it has been widely used for verification of program properties. Property verification often requires inter-procedural analysis, in which control and data flow are tracked across methods. At the core of inter-procedural analyses is the call graph, which establishes relationships between caller and callee methods. However, it is challenging to perform static analysis and compute the call graph of programs with dynamic features. Dynamic features are widely used in software systems; not supporting them makes it difficult to reason over properties related to these features. Although state-of-the-art research had explored certain types of dynamic features, such as reflection and RMI-based programs, serialization-related features are still not very well supported, as demonstrated in a recent empirical study. Therefore, in this paper, we introduce SALSA (STATIC ANALYZER FOR SERIALIZATION FEATURES), which aims to enhance existing points-to analysis with respect to serialization-related features. The goal is to enhance the resulting call graph's soundness, while not greatly affecting its precision. In this paper, we report our early effort in developing SALSA and its early evaluation using the Java Call Graph Test Suite (JCG).

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

Keywords: Java serialization, Java deserialization, Object marshaling and unmarshalling, Static analysis, Call graphs

ACM Reference Format:
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), July 23, 2020, Virtual, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3427761.3428343

FTfJP '20, July 23, 2020, Virtual, USA

ACM ISBN 978-1-4503-8186-4/20/07…$15.00
https://doi.org/10.1145/3427761.3428343

1 Introduction

Static analysis has been widely used for performing multiple types of program properties verification, such as vulnerability/bug detection, test case generation, and compiler optimizations [5, 7, 12, 15, 23, 36]. At the core of many of those analyses is the program's *call graph*, which establishes the relationships between callers and callees [16]. These call graphs are meant to model the possible program paths and it is a crucial element when performing inter-procedural analyses. However, many programming languages (including Java) contain dynamic features that introduce challenges for static program analysis.

Dynamic features are heavily used in contemporary software systems [21, 29] to link/load new class libraries, methods, and objects and extend the programs' functionalities. Therefore, ignoring such constructs leads to unsound call-graphs; they miss feasible runtime paths because they cannot infer the possible execution from the code [29, 30, 37]. To tackle this problem, previous literature explored certain classes of dynamic features, such as reflection features [6, 24, 25, 34], and programs with Remote Method Invocation (RMI) [33]. However, as demonstrated by Reif et al. [29, 30], one programming construct that has been left out from the programming analysis techniques is the support for handling serialization (and deserialization) of objects.

Object serialization (or *marshalling*) is the process of converting an object to an abstract representation, such as bytes, XML, JSON, etc. These representations are suitable for network transportation, storage, and inter-process communication. In Java, the serialization mechanism converts the objects' fields to a stream of bytes (i.e., it does not serialize code, only data). The receiver of a serialized object has to parse the abstract representation in order to reconstruct a new object. This reconstruction process is called *object deserialization* (or *unmarshalling*) [27].

Serialization-related features are used in many software systems [29]. It is one of the building blocks of Java RMI, Java Management Extensions (JMX), and other technologies. Therefore, adding support to this construct can help client analyses in reasoning over such programs. In particular, it enables finding reachable parts of the program via callback methods that are invoked during serialization and deserialization of objects [8].

In this paper, we present our early results in developing **SALSA (STATIC ANALYZER FOR SERIALIZATION FEATURES), an approach to statically analyze Java programs that contain serialization and deserialization in its code**. It is meant to complement existing call graph construction algorithms to improve their soundness with respect to serialization-related features. SALSA employs an iterative framework that constructs call graphs on-the-fly and iteratively refines them based on a set of assumptions about the code. When constructing the call graph, SALSA introduces *synthethic methods*, which are meant to model the behavior of the program during serialization/deserialization; indicating the possible callbacks that might be invoked during these processes. The contributions of this work are:

- an approach to improve call graphs' soundness with respect to serialization/deserialization features. It is agnostic to the underlying pointer analysis policy used to construct a call graph and is meant to complement them.
- a prototype implementation of the approach on top of WALA.
- an initial evaluation of the approach's soundness improvement using the Java Call Graph Test Suite [13].

2 Overview of the Java Serialization API

In Java, an object can be serializable into a stream of bytes as long as its class implements the java.io.Serializable interface. Only the object's state (field values) are serialized; its methods lie within the classpath of the receiver of the byte stream [32]. All non-static and non-transient fields in a class are serialized/deserialized by default. The ObjectOutputStream and ObjectInputStream classes from the java.io package can be used for serializing and deserializing objects, respectively. During serialization and deserialization, these classes may invoke ***callback methods***, which are a methods with certain signatures that serializable classes can declare to customize how their fields are serialized/deserialized [27].

Listing 1 has serializable classes examples[1], in which two of them have callback methods (lines 3-6, and 13-25). These methods take as arguments the current object input/output stream that can be used to read/write from/to the byte stream. Since the field a from MyList is *transient*, it is not serialized by default. Thus, its callback method writeObject() in MyList ensures that the elements in a are serialized in order. MyList's readObject() method reconstructs the array by first reading its size from the stream, allocating a with the right size, and finally reading each element from the stream [2].

The code snippet shown in Listing 2 serializes a Classroom object into a file. It first instantiates an ObjectOutputStream, passing to its constructor a FileOutputStream instance. Then, it calls writeObject() passing c1 as an argument, which serializes c1 as a byte stream and saves it in "class.txt".

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

[2]This sample implementation is similar to the one in java.util.ArrayList

```
1  class Student implements Serializable { protected String name; }
2  class TA extends Student{
3    private void readObject(ObjectInputStream s)
4      throws IOException, ClassNotFoundException { /* ... */ }
5    private void writeObject(ObjectOutputStream s)
6      throws IOException { /* ... */ }
7  }
8  class Classroom implements Serializable {
9    private int totalSeats; private MyList<Student> students;
10 }
11 class MyList extends AbstractList<Student> implements Serializable{
12   private transient Student[] a; private int size;
13   private void readObject(ObjectInputStream s)
14     throws IOException, ClassNotFoundException {
15     s.defaultReadObject();
16     a = (Student[]) new Object[size];
17     if (size > 0) {
18       for (int i = 0; i < size; i++) a[i] = (Student) s.readObject();
19     }
20   }
21   private void writeObject(ObjectOutputStream s)
22     throws IOException {
23     s.defaultWriteObject();
24     for (int i = 0; i < size; i++) s.writeObject(a[i]);
25   }
26 }
```

Listing 1. Examples of Serializable classes

```
Classroom c1 = new Classroom(30,
  new MyList<>(new Student[]{new Student("John"), new TA("Jane")}));
FileOutputStream f = new FileOutputStream(new File("class.txt"));
ObjectOutputStream out = new ObjectOutputStream(f);
out.writeObject(c1);
```

Listing 2. Object serialization in Java

Listing 3 has a code snippet that deserializes this object from the file. This code creates an ObjectInputStream instance. Then, it invokes the method readObject(), which parses the stream of bytes and returns an object. The returned object is finally casted to the Classroom class type.

```
FileInputStream fs = new FileInputStream(new File("class.txt"));
ObjectInputStream in = new ObjectInputStream(fs);
Classroom c2 = (Classroom) in.readObject();
```

Listing 3. Object deserialization in Java

Figure 1 contains a sequence diagram with the major methods invoked during the execution of Listings 2 and 3. Classes with a gray background are part of the Java's API, whereas the ones with a white background are application classes. As shown in this diagram, the callback methods are (indirectly) called by the ObjectStreamClass via reflection (marked in red dashed arrows). During serialization and deserialization, both writeObject and readObject from MyList are invoked. Since one element in a is of type TA, the writeObject and readObject methods from TA are also invoked via reflection.

3 Approach Overview

From the examples shown in Section 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**

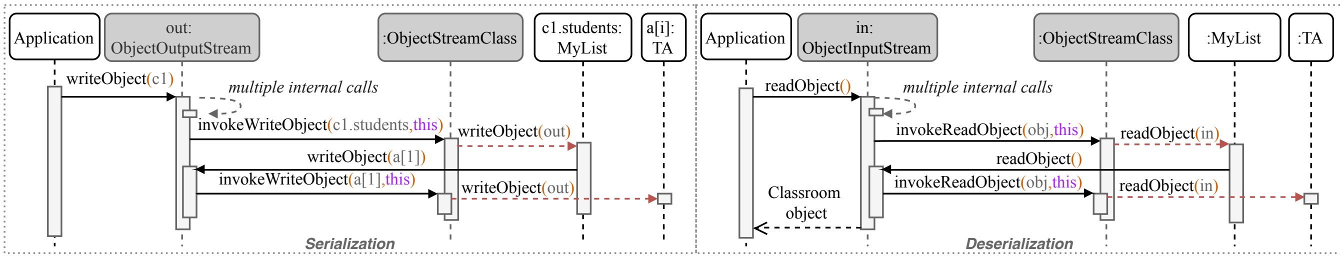

Figure 1. Method calls during serialization/deserialization of objects

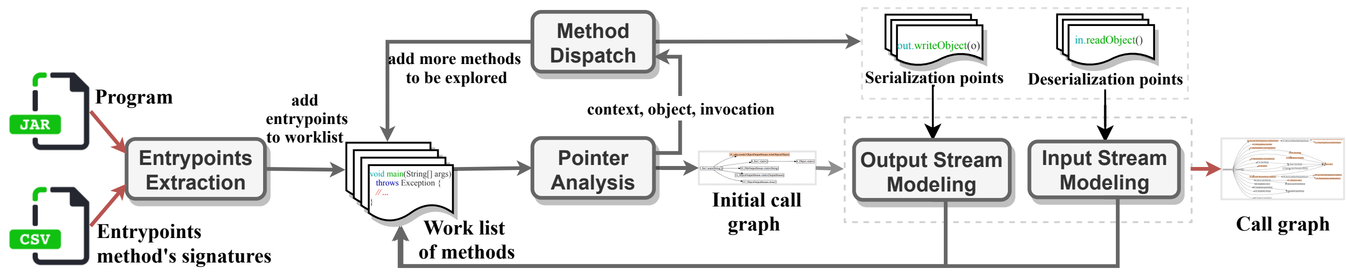

Figure 2. Overview of the approach employed by SALSA

that are invoked during the serialization/deserialization; and (ii) the **fields within the class can be allocated in unexpected ways**.

When deserializing an object, which actual callback methods invoked at runtime depends on the byte stream whose contents is unknown during static analysis. For instance, if the code snippet in Listing 2 had only the student "John" in the list (line 2), then the calls to readObject/writeObject methods in TA would not be made.

Existing pointer analysis policies leverage on allocation instructions (new T()) within the program to infer the possible runtime types for objects [4, 14, 17, 18, 20, 22, 31, 35]. 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 serialized 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.

To handle these challenges, we make these assumptions:

(1) **There is no dynamic loading of remote classes**. Thus, only the classes in the classpath are available for serialization (closed-world assumption) [26];
(2) **All fields in serializable classes are not null**. They can be allocated with any type that is safe. This assumption ensures that we can soundly infer the possible targets for invocations within callback methods made via inner fields (e.g., lines 18 and 24 in Listing 1).
(3) **All type refinements (downcasts) are safe**. Hence, they can be used to infer the possible callback methods invoked during the serialization/deserialization and points-to sets for fields within serializable classes. This assumption is crucial to improve the call graph's soundness while not greatly degrading its precision since many classes in the classpath implement the java.io.Serializable interface.

To support serialization-related features we developed SALSA. It employs an iterative approach for building the program's call graph [16]. The approach 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 policy; ② An iterative refinement of the initial call graph by applying the assumptions aforementioned. In the next subsections, we first present definitions for relevant concepts to make the work understood by a broader audience. Next, we explain how SALSA enhances existing pointer analysis policies to support serialization-related features by performing call graph refinement via code modeling.

3.1 Definitions

Below we define concepts needed for understanding our solution formulation subsequently described. We use similar terminology as [37].

DEFINITION 1. **Scope**: Each instruction i enclosed in a method m in a program under analysis has a *scope*. The scope is based on where m is declared and it can either be *application*, *extension* (code from libraries/APIs), or *primordial* (Java's standard API classes).

Definition 2. **Serialization Points**: Instructions i within the *application* scope that invokes ObjectOutputStream's writeObject(Object) are serialization points; they convert an object into a stream of bytes.

Definition 3. **Deserialization Points**: Instructions i within the *application* scope that invokes ObjectInputStream's readObject() are deserialization points; they reconstruct an object from a byte stream.

Definition 4. **Method Contexts**: Each method m in the program has an associated context c, where $contexts(m)$ track all the contexts that have arisen for m. A context c is an abstraction of the program's state.

Definition 5. **Pointer**: A variable x in a method m at a context c has an associated abstract pointer $p = \langle x, c\rangle$.

Definition 6. **Points-to sets**: A points-to set $pt(p)$ tracks the variables or heap locations to which the pointer p can point to. Every variable x in a context c has an associated points-to set $pt(\langle x, c\rangle)$.

Definition 7. **Worklist of Methods**: Salsa maintains a worklist $\mathcal{W}$ which tracks the methods m under a context c that have to be traversed ($\langle m, c\rangle \in \mathcal{W}$).

Definition 8. **Synthetic Methods**: Salsa employs synthetic methods $m_s \in M_s$ to model the possible method calls during serialization/deserialization. Thus, the program's call graph includes "fake" nodes computed from these synthetic methods m_s under a context c.

3.2 Phase 1: Initial Call Graph Construction

The first step in our approach is to ***extract the program's entrypoints***, which are the methods that start the program's execution. We use the main() methods as entrypoints by default. However, client analyses can provide a CSV file with method signatures for entrypoints (useful for Web applications/services written in Java which can process requests from many entrypoint methods). The result of this step is a list of entrypoint methods m added to our worklist $\mathcal{W}$. Since the worklist tracks methods within a context, the entrypoints methods are assigned a global context [37].

Starting from the entrypoint methods identified, Salsa constructs an ***initial (unsound) callgraph*** using the underlying algorithm selected by the client analysis (e.g., n-CFA, etc). 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) [9]. Each instruction in this IR is visited following the rules by the underlying pointer analysis algorithm. We point the reader to the work by Sridharan et al. [37] which provides a generic formulation for multiple points-to analysis policies.

When visiting instance invocation instructions (i.e., x = o.g(a_1,a_2,...,a_n)) in a method m, the static analysis computes the possible dispatches (call targets) for the method g as follows:

$$\text{targets} = dispatch(pt(\langle o, c\rangle), g)$$

The dispatch mechanism takes into account the current points-to set for the object o at the current context c. If the invocation instruction occurs at a *serialization or deserialization point*, then the *dispatch* function implemented by Salsa 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 created at this phase *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* [37]. 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.

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

3.3 Phase 2: Call Graph Refinement

In this phase, we take as input the current call graph g which contains as nodes actual methods in the application and synthetic methods created by Salsa in the previous phase. At this phase, Salsa adds instructions to these synthetic models by applying the assumptions mentioned at the beginning of this section, described in detail as follows.

3.3.1 Modeling Object Serialization. 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 (the **foreach** in line 10), we iterate over all non-static fields f from the class t and compute their points-to sets. 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.

After adding all the needed instructions to the synthetic method m_s, we re-add the synthetic method to Salsa's worklist (as depicted in Figure 2).

3.3.2 Modeling Object Deserialization. Since multiple classes in a classpath (e.g., Java's Swing classes) implement

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
foreach instruction in I do
    o_i ← argument(1,instruction)
    c ← context(instruction)
    m_s ← target(instruction)
    foreach t ∈ pt(⟨o_i, c⟩) do
        addTypeCast(m_s,t)
        if t has a writeObject(ObjectOutputStream) callback then
            addInvoke(m_s, t.writeObject)
        end
        foreach f ∈ fields(t) do
            foreach fieldType ∈ pt(⟨o_i.f, c⟩) do
                if fieldType has writeObject(ObjectOutputStream)
                 then
                    addGetField(m_s, f)
                    addTypeCast(m_s, fieldType)
                    addInvoke(m_s, fieldType.writeObject)
                end
            end
        end
    end
    addToWorkList(m_s,c)
end
```

the java.io.Serializable interface, objects received from a source stream can be of any of these classes. Thus, there is a high amount of possible calls that would be erroneously included in the resulting call graph. To tame this complexity, we assume that only the classes in the classpath are serialized, all their instance fields are non-null, and downcasts are safe when modeling the serialization mechanism. Algorithm 2 contains the steps performed in this modeling.

We first traverse the def-use chains [1] of the caller's IR to find any downcasts for the returned deserialized object:

$$o_{ret} = \text{in.readObject()}$$
$$\vdots$$
$$x = (\text{ClassType})\ o_{ret}$$

For each downcast type, we add an allocation instruction into m_s followed by an invocation to the type's readObject() callback method (if any exists). Subsequently, we iterate over all instance fields of the type and compute the possible serializable classes that are type-safe for the field. For each possible type safe, we add a field allocation. Then, if the possible type has a callback method, we add two more instructions into m_s: a cast to the possible type and an invocation to the callback. After adding the aforementioned instructions to m_s, the synthetic method is re-added to the worklist.

3.4 Running Example

Figure 3 partially shows the call graph SALSA computes for the Listing 2. To build this call graph, SALSA computes the initial call graph (using 0-1-CFA in this example). The initial call graph contains one synthetic method modeling ObjectOutputStream's writeObject(...) called at main. The synthetic method is initialized without any instructions (Phase 1). In

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 /* re-added to the
        worklist                                              */
foreach instruction in I do
    c ← context(instruction)
    m_s ← target(instruction)
    o_ret ← argument(1,instruction)
    foreach t ∈ downcasts(o_ret) do
        o_i ← addAllocation(m_s,t)
        if t has a readObject(ObjectInputStream) callback then
            addInvoke(m_s,t.readObject)
        end
        foreach f ∈ fields(t) do
            foreach type ∈ possibletypes(f) do
                addAllocation(m_s,o_i.f, type)
                if type has readObject(ObjectInputStream) then
                    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
```

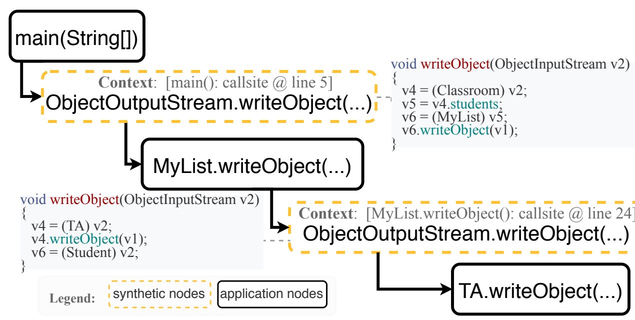

Figure 3. Computed call graph for Listing 2

Phase 2, SALSA refines the initial call graph by adding instructions to this first synthetic method. The added instructions include a possible call to MyList's writeObject. After enriching the synthetic method with instructions, SALSA adds the synthetic method back again to the worklist for further analysis by the *pointer analysis component* and *dispatch mechanism*. After visiting all instructions from the synthetic node, there is a new serialization point at it (as highlighted in yellow). Thus, the dispatch mechanism adds a new node to the call graph corresponding to a second synthetic model which arises at line 24 in Listing 1. This second synthetic method is added to the call graph with no instructions. This synthetic method is then refined by adding instructions to it which indicates a possible invocation to the callback method from the TA class. At this stage, no more refinements are needed (since no more serialization points are uncovered at the synthetic method introduced).

Table 1. Results from running the test cases from JCG

Ser1	Ser2	Ser3	Ser4	Ser5	Ser6	Ser7	Ser8	Ser9
✓	✓	✓	✓	✓	✗	✗	✗	✗

4 Early Results

We developed a prototype for Salsa in Java using IBM's T. J. Watson Libraries for Analysis (**WALA**) [19]. In this section, we discuss initial results for the research question:

RQ *Does the approach improve in terms of soundness with respect to serialization features?*

To answer this question, we run Salsa with the Java Call Graph Test Suite (JCG) [13, 29, 30]. This test suite contains nine test cases (`Ser1-9`) with serialization or deserialization in it. Each test case is a Java program with annotations that indicate the expected targets for a method call. Table 1 reports the test cases that Salsa passed (✓) and the ones it failed (✗). The computed call graphs are released at our repository https://github.com/SoftwareDesignLab/Salsa.

Salsa passed 5 out of 9 test cases. The test cases Ser6-9 failed because they involved callback methods that Salsa's prototype currently does not support (i.e., readResolve, validateObject, and writeReplace). Adding support to these callbacks is part of our ongoing efforts in improving.

Although Salsa did not pass all test cases in the JCG test suite, it is important to highlight that existing call graph construction algorithms only passed either 1 test case ($Soot_{RTA}$ and $Soot_{CHA}$) or 5 test cases ($OPAL_{RTA}$) [29]. Even then, they use imprecise call graph construction algorithms, Class Hierarchy Analysis (CHA) [10] and Rapid Type Analysis (RTA) [3] which creates large and imprecise call graphs (in terms of nodes and edges) because they only rely on static types when computing the possible targets of a method invocation. Salsa keeps a balance between improving soundness while not greatly affecting the call graph's precision.

5 Future Work

We intend to improve Salsa concerning the following:

• ***Handle cases in which classes explicitly declare which fields should be serialized***: In Java, a developer can define the fields to be serialized in two ways: *implicitly* (all the non-transient and non-static fields are serialized by default); or *explicitly* by declaring an extra field (serialPersistentFields), that indicates names and types of the serializable fields. Salsa currently assumes that the classes declare the serializable fields implicitly.

• ***Provide support for serialization via the Externalizable interface***: Unlike the Serializable interface which use Java's serialization protocol [27], the Externalizable interface has its own callback methods and the application classes have to implement the serialization process themselves.

• ***Model other callback methods***(e.g., validateObject()) [27].

Moreover, we will evaluate Salsa using real software systems. We will verify whether Salsa is scalable to realistic programs. We will also inspect to what extent the approach affects the call graph's precision (i.e., how many spurious paths are added to the call graph).

6 Related Work

Many works explored the problem of performing pointer analysis of programs [4, 14, 17, 18, 20, 22, 31, 35]. These approaches focus on computing over- or under-approximations in order to improve one or more aspects of the analysis, such as its soundness, precision, performance, and scalability. In this paper, we focus on aiding points to analysis in handling by serialization-related features in a program. Previous research on static analysis also explored the challenges involving supporting reflection features [6, 24, 25, 34]. These approaches involve making certain assumptions when performing the analysis, in order to create analyses that are not overly imprecise. Sharp and Rountev discussed an approach to statically analyze RMI-based programs, which requires reasoning over client and server code and their inter-process communication via objects/messages [33]. In the past few years, there was a spike of vulnerabilities associated with deserialization of objects [8]. Thus, existing works also studied vulnerabilities rooted at untrusted deserialization vulnerabilities [11, 28]. Pele *et al.* [28] conducted an empirical investigation of deserialization of pointers that lead to vulnerabilities in Android applications and SDKs. Dietrich et al [11] demonstrated how seemingly innocuous objects trigger vulnerabilities when deserialized, leading to denial of service attacks. There is a line of research that explored call graph's soundness of Java (or JVM-like) programs [2, 29, 30]. In particular, recent empirical studies [29, 30] show that although serialization-related features are widely used, they are not well supported in existing approaches. Currently, to the best of our knowledge, we could not find an approach that aims to enhance existing points-to analysis to support serialization-related features.

7 Conclusion

We presented Salsa, an approach to support the static analysis of serialization-related features in Java programs. By applying assumptions, Salsa adds synthetic nodes into a previously computed call graph to improve its soundness with respect to serialization-related features. We provided initial results concerning to which extent Salsa can improve call graphs' soundness by running Salsa against test cases from the Java Call Graph Test Suite (JCG).

Acknowledgments

This work was partially funded by the US National Science Foundation under grant number CNS-1816845.

References

[1] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. 1986. Compilers, principles, techniques. *Addison wesley* 7, 8 (1986), 9.

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

[3] 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

[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 (ECOOP 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. https://doi.org/10.4230/LIPIcs.ECOOP.2019.11

[5] M. Beller, R. Bholanath, S. McIntosh, and A. Zaidman. 2016. Analyzing the state of static analysis: A large-scale evaluation in open source software. In *23rd International Conference on Software Analysis, Evolution, and Reengineering*. 470–481. https://doi.org/10.1109/SANER.2016.105

[6] 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

[7] Jesús Mauricio Chimento, Wolfgang Ahrendt, and Gerardo Schneider. 2018. Testing Meets Static and Runtime Verification. In *Proceedings of the 6th Conference on Formal Methods in Software Engineering* (Gothenburg, Sweden) *(FormaliSE '18)*. Association for Computing Machinery, New York, NY, USA, 30–39. https://doi.org/10.1145/3193992.3194000

[8] 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*. 7–12. https://doi.org/10.1145/2771284.2771286

[9] 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

[10] 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

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

[12] V. D'Silva, D. Kroening, and G. Weissenbacher. 2008. A Survey of Automated Techniques for Formal Software Verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 27, 7 (2008), 1165–1178. https://doi.org/10.1109/TCAD.2008.923410

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

[14] 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

[15] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. 1987. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 9, 3 (1987), 319–349. https://doi.org/10.1145/24039.24041

[16] 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

[17] 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

[18] 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

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

[20] 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

[21] 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 Press, 507–518. https://doi.org/10.1109/ICSE.2017.53

[22] 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

[23] Li Li, Tegawendé F. Bissyandé, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein, and Le Traon. 2017. Static analysis of android apps: A systematic literature review. *Information and Software Technology* 88 (2017), 67 – 95. https://doi.org/10.1016/j.infsof.2017.04.001

[24] 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

[25] 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

[26] 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* (Tsukuba, Japan) *(APLAS'05)*. Springer-Verlag, Berlin, Heidelberg, 139–160. https://doi.org/10.1007/11575467_11

[27] Oracle. [n.d.]. Java Object Serialization Specification (version 6.0). https://docs.oracle.com/javase/8/docs/platform/serialization/spec/serialTOC.html. (Accessed on 05/24/2020).

[28] 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.

[29] 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)*. Association for Computing Machinery, New York, NY, USA, 251–261. https://doi.org/10.1145/3293882.3330555

[30] 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

[31] 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

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

[33] 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

[34] 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

[35] 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

[36] 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*. 1053–1068. https://doi.org/10.1145/2048066.2048145

[37] Manu Sridharan, Satish Chandra, Julian Dolby, Stephen J Fink, and Eran Yahav. 2013. Alias analysis for object-oriented programs. In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*. Springer, 196–232. https://doi.org/10.1007/978-3-642-36946-9_8