SALSA: Static Analysis of Serialization Features

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

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

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

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

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 callgraphs; 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** (**S**tatic **A**nalyst for **S**er**i**alization **F**eatures), 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 **synthetic 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 [15].

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 class Student implements Serializable {
  2  protected String name;
  3  protected int age;
  4  private int grade;
  5  public Student(String name, int age, int grade) {
    6      this.name = name;
    7      this.age = age;
    8      this.grade = grade;
  9  }
 10  public String getName() { return name; }
 11  public int getAge() { return age; }
 12  public int getGrade() { return grade; }
}
```

```
1 Listing 1. Examples of Serializable classes
```

```
1 class TA implements Serializable {
  2  private String name;
  3  private int age;
  4  public TA(String name, int age) {
    5      this.name = name;
    6      this.age = age;
  7  }
 8  public String getName() { return name; }
 9  public int getAge() { return age; }
}
```

```
1 class Classroom implements Serializable {
  2  protected String name;
  3  public Classroom(String name) {
    4      this.name = name;
  5  }
  6  public String getName() { return name; }
}
```

```
1 Listing 2. Object serialization in Java
```

```
1 File file = new File("class.txt");
```

```
1 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**

---

\(^1\) 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.
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
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 \( W \). 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 \( (m, c) \in W \) 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., \( x = \text{obj}(a_1, a_2, ..., a_n) \)) in a method \( m \), the static analysis computes the possible dispatches (call targets) for the method \( g \) as follows:

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

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 ObjectOutputStream and ObjectInputStream, 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 ObjectOutputStream/ObjectInputStream 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 \( m_t \) a 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 \( m_t \) 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 \( m_t \), 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
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 serializable, 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:

\[
\text{\(x = \text{(ClassType) \text{\(\sigma_{\text{ret}}\)}\)}
\]

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 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).
We intend to improve 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 (SootRTA and SootCHA) or 5 test cases (OPALRTA) [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


