Finding Architectural Flaws in Android Apps Is Easy

Radu Vanciu  Marwan Abi-Antoun
Department of Computer Science, Wayne State University
{radu, mabiantoun}@wayne.edu

Abstract
Mobile devices store confidential information. As a result, security vulnerabilities such as information disclosure in mobile apps can have serious consequences. To build secure apps, developers are expected to follow security policies that are described only informally. Some policies target architectural flaws, rather than coding defects, and are not easily checked or enforced with existing tools that focus on low-level coding defects.

Scoria is a prototype tool that allows architects to write security policies as machine-checkable constraints that are executed against a program abstraction that includes a hierarchy of abstract objects with dataflow communication edges. Using Scoria, architects reason not only about the presence or absence of communication, but also about object provenance, object hierarchy and reachability. We show how Scoria can find information disclosure in an open-source Android application.

1. Introduction
We propose a semi-automatic approach, Scoria, to find architectural flaws such as information disclosure. One requirement of security analyses that find architectural flaws is to use a runtime architecture [4]. Previous work approximates a runtime architecture by a sound, hierarchical, Ownership Object Graph (OOG) that is statically extracted from code with annotations [2]. The OOG has nodes that represent abstract objects and groups of abstract objects, and edges that represent relations between objects. In an object hierarchy, an object does not have child objects directly. Instead, an object contains groups of objects (domains). A points-to edge is a relation between two abstract objects due to a field reference. A dataflow communication between two abstract objects is due to a method invocation, field read or field write. The edge label is an abstract object that the dataflow communication refers to. Similarly, a creation edge is due to an object allocation expression, where the edge label refers to an abstract object.

Scoria uses a security graph that is an OOG enriched with security properties and queries (SecGraph). Architects deepen their understanding of a runtime structure as-needed and assign values for the security properties of abstract objects and edges using queries. Using queries and constraints, the architects use Scoria to convert informally-specified security policies into machine-checkable constraints.

2. Description
We will present how an architect uses Scoria to find information disclosure. The main contribution of Scoria are queries that capture the thought process of how an architect can reason about the communication of objects in a runtime architecture. For example, to find information disclosure, the architect assigns security properties to objects and finds a confidential object that flows into an untrusted destination. One consequence of representing dataflow as an object is that an architect can use object identity to reason about the information content available from an object. The dataflow object may not be confidential, but a confidential object could be reachable from it or be contained within its substructure.

Scoria automatically considers object ancestors and descendants, transitive communication, and object reachability. Such information allows the architect to reason about the object provenance. That is, an architect can reason about “what” object a dataflow communication refers to, and “how” other objects are using the same dataflow object. Scoria also enables architects to query indirect communication through object hierarchy and reachability. Architects may miss such communication if they were to reason based on a diagram alone, because it requires computing transitive information.

The demonstration starts with examples of queries that architects can use to assign values to properties of objects in the SecGraph. Next, architects use constraints based on object provenance and indirect communication queries to find architectural flaws as suspicious dataflow communication edges. A SecGraph also provides traceability information such that architects can trace from a suspicious edge directly to the code and investigate the vulnerability.
void insecureIntents() {
    Property[] snkP = {TrustLevelType.Untrusted};
    Property[] flwP = {IsConfidential.True};
    g.assignProperty(snkP, new InstanceOf("Intent"));
    g.assignProperty(flwP, 
        new IsChildOf("AccountInformation","String"));
    Set<SecEdge> se = g.getFlowSink(snkP, flwP);
    Assert.assertTrue(se.size()>0)
    displayWarnings(se);
}

Figure 1. A constraint implemented as a JUnit test to find
information disclosure in objects of type Intent

3. Example

We use Scoria to find information disclosure in UPMA, an
open source Android application for managing passwords.
During the demonstration, we will assume that the OOG has
been already extracted in order to focus on the process of
assigning security properties and writing constraints.

Android security policy dictates that confidential infor-
mation should not be disclosed to objects of type Intent,
since other applications can access it. We implement this
policy as the insecureIntents constraint (Fig. 1) that is then
executed against the UPMA SecGraph. First, the constraint
assigns the Untrusted property to all objects of type Intent.
Objects of the same type can have different roles. Sco-
ria allows the architect to assign the IsConfidential prop-
erty to those objects of type String that have a parent of
type AccountInformation. Next, the constraint finds an
information disclosure in the ViewAccountDetails class
through a creation edge (Fig. 3). The password is ex-
posed indirectly being a child of the AccountInformation
object that is referred by the ViewAccountDetails object.
In addition, Scoria finds a second vulnerability in the
AccountsList class where the unencrypted password is
disclosed to the Android clipboard through a dataflow edge.
Since the clipboard is accessible to any other app, passwords
stored by UPMA are vulnerable to eavesdropping.

4. Implementation

Scoria is based on a typechecker, an extraction analysis,
and a constraint checker. The typechecker and the extrac-
tion of OOG with points-to edges were previously demon-
strated [1]. In this demonstration, we focus on extracting
dataflow edges [5] and on implementing security policies
as constraints. The tools can analyze any Java code, which
makes them suitable to analyze Android apps.

The extraction analysis is implemented using the Crystal
framework [3] as an Eclipse plugin. Constraints are imple-
mented as SecGraph queries, such that architects can write
and execute constraints as JUnit tests. From the query re-
results, Scoria can trace to AST nodes, such that the architect
locates the vulnerability in the code. To debug queries, Sco-
ria also includes a visualization of the SecGraph (Fig. 2).

Figure 2. A fragment of the OOG that shows an indirect
communication of password:String to intent:Intent, and a
direct communication of password:String to cm:ClipboardManager

Figure 3. UPMA code that introduce vulnerabilities

References

Extraction of Sound Hierarchical Representations of Runtime
Object Graphs (Tool Demonstration). In OOPSLA, pages 743–
744, 2008.
mance Analysis of Hierarchical Runtime Architectural
[3] PLAID Research Group. The Crystal Static Analysis Frame-
2004.