Analyzing Sandboxed Interpreters with Abstract Interpretation

Phillip Mates
University of Utah
philip.mates@utah.edu
50 S. Central Campus Drive, Room 3190, SLC, UT 84112
Advisor: Matthew Might
Undergraduate

Extended Abstract

0.1 Problem and Motivation:
The Android platform provides a coarse-grained per-application permission policy. While this approach works in general, applications that contain multiple subprograms would benefit from more fine-grained permission guarantees. For instance, an advertisement-serving GPS app requires both Internet and Location permissions, but provides no guarantee that your location won’t be leaked. Another example is when applications use embedded interpreters to enable execution of user-defined plugins. Interpreting code at runtime in an application that’s granted myriad permissions has the potential to introduce information leaks. This makes it important to be able to prove that an embedded interpreter cannot make use of certain permissions available to its host application.

To ensure that interpreters are properly sandboxed, we implement an abstract interpreter for the Dalvik bytecode, capable of soundly approximating an Android application’s state space. The motivations for this research are two-fold: We would like to provide a useful tool for statically verifying that an embedded interpreter does not abuse the permissions granted to its host applications. Secondly, this is a solid step toward verifying the absence of malware and conducting deep, interprocedural optimization of object-oriented code.

0.2 Background and Related Work:
Permissions granted to an Android application are defined in a Manifest File and set at installation. These permissions specify the application’s ability to access sensitive system calls and other applications’ interfaces. For an in-depth description of the Android permission model, we refer the reader to [1].

The inflexibility of this model has lead researchers to develop numerous refinements, both dynamic [5] [2] and static [4] [3]. Dynamic approaches require additional CPU services on the mobile device to monitor data flow [2] or re-route permission requests through a gatekeeper application [5]. Static approaches make use of Java static analysis suites to uncover potential permission violations and require either the original Java source code [4] or decompilation [3].

Our approach also looks to statically enforce permission properties but only requires the packaged application, since we target the Dalvik bytecode.

0.3 Approach and Uniqueness:
We have implemented two interpreters: the Lambda interpreter and the Lambad interpreter, both embedded in identical host Android applications. The Lambda interpreter operates on a vanilla $\lambda$-calculus language that is extended with numbers and simple arithmetic operations. The Lambad interpreter is identical but secretly extends its environment to include a publishLocation function freely available in the hosting application. To show that code interpreted by the Lambad can introduce privacy leaks while the Lambda interpreter cannot, we’ve developed an abstract interpreter to soundly approximate states reachable by each embedded interpreter.

Our abstract interpreter operates over Dalvik bytecode and was built using methodologies described in [7]. We incrementally tweaked a concrete CESK machine until the infinite structures and values were made finite. The abstract domain for primitives and objects is flat and we employ an abstract garbage collector [6] for improved precision and run-time complexity. The result is a highly adjustable k-CFA style analyzer capable of soundly approximating Android applications.

For simplicity, we currently do not handle listeners, intents, or threads. Library function calls are simulated by returning an abstract object of the given function’s return type.

This approach to abstract interpretation is novel because it takes techniques such as abstract garbage collection [6] or tunable interpreters [7], which have previously been implemented on simple toy languages, and applies them to an industrial strength language.

0.4 Results and Contributions:
Our research is still in its early stages. We’ve implemented an abstract interpreter that is capable of analyzing small segments of Android code, such as the recursive Fibonacci function seen in Fig 1. We are very close to being able to analyze the Lambda and Lambad embedded interpreters, but our analysis is still too imprecise.

While a tool able to prove an embedded interpreter is incapable of malicious activity will be useful, we see this as the first of many applications for abstract interpretation in the Android development environment. There are still numerous theoretical results in abstract interpretation waiting to be put to use in an industrial setting.

Project source code available at:
https://github.com/phillipm/dalvik-abstract-interpreter

Interactive version of Fig 1:
http://eng.utah.edu/~mates/files/fib/

Figure 1. A state space graph of a Fibonacci function
References


