Symbolic Execution of Dalvik Bytecode
EECS 450 Class Project
Midterm Presentation

Jedidiah McClurg    Jonathan Friedman    William Ng
Mentor: Vaibhav Rastogi

Northwestern University

April 30, 2012
Android is a popular smartphone OS designed by Google
Android is designed for security

- Linux-based OS in which each application runs on a VM in its own process
- Each application has unique user ID, preventing them from interacting maliciously
- Explicit user permission is required for apps to access devices

Android’s virtual machine is called Dalvik
Dalvik bytecode is similar to Java bytecode, with one major difference being its register-based (rather than stack-based) architecture
Problem Statement

- Android can still be vulnerable to several types of malware [3]
- Some attacks take advantage of user’s haste or carelessness
- A specific attack of this form which we seek to address is the following:

Malware strategy

Obtain blanket permission to use SMS messaging, and then incur messaging fees or subscribe to “premium” services without the user’s knowledge

- This type of attack can be detected using various program analyses [1]
  - Dynamic analysis – this is fast, but focuses on a limited number of program execution paths
  - Static analysis – this can cover all execution paths, but can be very slow as program complexity increases
Symbolic Execution

A hybrid solution: symbolic execution
Instead of running an application on the VM with *concrete* input/output, we can execute it using *symbolic* IO:

Concrete Execution

```plaintext
x = readint();  // wait for int, e.g. "5"
b = 10 + x;
b *= 2;
return b;
// result: 30
```

Symbolic Execution

```plaintext
x = readint();  // x is now symbol A
b = 10 + x;
b *= 2;
return b;
// result: 2*(10+A)
```

- This approach “executes” the program in a limited number of symbolic paths, so approximates the speed of dynamic analysis while covering more actual paths
- It focuses on collecting *constraints* rather than *values*, so it’s faster than static analysis
Detecting Threats via Symbolic Execution

- Assuming we are trying to detect calls to unexpected SMS messaging, consider the following code:

```c
y = 5;
x = getchar();
if(2*x == y - 1) {
    sendSMS(evilPhoneNumber, "subscribe");
    return 1;
}
return 0;
```

- Let’s say in our concrete execution, we got the character 'z' from the keyboard... so, \( x = 122 \) meaning the body of the if statement will be skipped (overlooked threat!)

- Symbolic execution would find that the body is evaluated iff \( 2x = 5 - 1 \), i.e. \( x = 2 \) (potential threat found!)
Our symbolic execution system works at the Dalvik assembly code level.

Android applications are distributed as APK files, which can be decompiled using apk-tool/smali.

```asm
.class public Lcom/myspace/android/MySpace;
.super Landroid/app/Activity;
.source "MySpace.java"

.method private openHomePage()V
    .locals 2
    .prologue
    new-instance v0, Landroid/content/Intent;
    const-class v1, Lcom/myspace/android/pages/HomePage;
    invoke-direct {v0, p0, v1},
        Landroid/content/Intent;-><init>(Landroid/content/Context;
            Ljava/lang/Class;)V

    .local v0, myIntent:Landroid/content/Intent;
    const/4 v1, 0x0
```

The instructions (and structural directives provided by apk-tool/smali) are parsed into an abstract syntax tree (AST).
In order to generate the symbolic constraints as we process the assembly code, we need to have a formal semantics for Dalvik bytecode.

Dalvik instructions are fairly low-level, so it is relatively straightforward to develop a formal semantics for Dalvik bytecode [2].

Using a similar approach, we are building a structural operational (i.e. *compositional*) semantics:

\[
\begin{align*}
\langle \text{nop}, (H, R, pc) \rangle &\longrightarrow (H, R, pc + 1) \\
\langle \text{add-int } a\ b\ c, (H, R, pc) \rangle &\longrightarrow (H, R[a \mapsto [b] + [c]], pc + 1) \\
\langle S_1, (H, R, pc) \rangle &\longrightarrow (H', R', pc') \\
\langle S_1\ S_2, (H, R, pc) \rangle &\longrightarrow \langle S_2, (H', R', pc') \rangle
\end{align*}
\]
The parser and semantics are almost finished, and then we can implement the symbolic execution engine.

This module will load the AST representation of the application and symbolically execute it with respect to the operational semantics.

To make this more manageable, we will instrument the bytecode (i.e. AST) with symbolic functionality, instead of building an interpreter from scratch.

The Dalvik VM (possibly via the Android emulator) will then be used to perform the symbolic simulation by simply running the recompiled APK file and generating a logfile with symbolic constraints for each instruction of interest.

Finally, we will implement a checker by sending the constraints and the properly-structured assertion to a model checker.
Overall System Architecture

The box on the right is the “core” of the symbolic simulator:

- The **Parser** loads the Dalvik code into a data structure
- The **Symbolic Code Generator** instruments the parsed code using the symbolic constraint propagation rules
- The **Constraint Solver** checks the property with respect to constraints generated by running the instrumented application

Jedidiah McClurg, Jonathan Friedman, William Ng
Symbolic Execution of Dalvik Bytecode
Security issues can arise in Android due to unexpected use of user information.

These types of information leaks can be detected by program analysis.

We seek to detect a simple set of events efficiently by symbolic simulation of Dalvik bytecode.

This type of functionality could be integrated quite smoothly into the Dalvik VM itself, offering a higher level of security.
