Instrumenting Android and Java Applications as Easy as abc

Steven Arzt, Siegfried Rasthofer, and Eric Bodden

Secure Software Engineering Group,
European Center for Security and Privacy by Design (EC SPRIDE),
Technische Universität Darmstadt, Germany
{steven.arzt,siegfried.rasthofer,eric.bodden}@ec-spride.de

Abstract. Program instrumentation is a widely used mechanism in different software engineering areas. It can be used for creating profilers and debuggers, for detecting programming errors at runtime, or for securing programs through inline reference monitoring.

This paper presents a tutorial on instrumenting Android applications using Soot and the AspectBench compiler (abc). We show how two well-known monitoring languages –Tracematches and AspectJ– can be used for instrumenting Android applications. Furthermore, we also describe the more flexible approach of manual imperative instrumentation directly using Soot's intermediate representation Jimple. In all three cases no source code of the target application is required.

Keywords: Android, Java, Security, Dynamic Analysis, Runtime Enforcement.

1 Introduction

According to a recent study [1], Android now has about 75% market share in the mobile-phone market, with a 91.5% growth rate over the past year. With Android phones being ubiquitous, they become a worthwhile target for security and privacy violations. Attacks range from broad data collection for the purpose of targeted advertisement, to targeted attacks, such as the case of industrial espionage. Attacks are most likely to be motivated primarily by a social element: a significant number of mobile-phone owners use their device both for private and work-related communication [2]. Furthermore, the vast majority of users installs apps containing code whose trustworthiness they cannot judge and which they cannot effectively control.

One approach to combat such threats is to augment Android applications obtained from arbitrary untrusted sources with additional instrumentation code. This code alters the behaviour of the target application and can thus enforce certain predefined security policies such as disallowing data leaks of confidential information. Since the instrumentation code runs as an integrated part of the target application, it has full access to the runtime state, thereby avoiding the imprecisions that usually come with static analysis approaches [3–5]. It has full

A. Legay and S. Bensalem (Eds.): RV 2013, LNCS 8174, pp. 364–381, 2013.

access to environment information, user inputs, and external resources. Policy violations can be captured as they actually occur, thus minimizing the number of false alarms. Furthermore, it has also the advantage that the underlying Android framework does not have to be changed at all, as done so by [6, 7].

In many cases, the source code of the target application is not available. Therefore, a mechansim for conveniently analyzing and instrumenting binary applications is required. Soot [8] and the abc compiler [9] for AspectJ both support Android bytecode, both as input and as output for the instrumented application. In this paper, we will give an overview of the two tools, explain how to integrate instrumentation code on various layers of abstraction, and illustrate the mechanisms using examples. Though this paper focuses on the Android platform, many of the tools and concepts presented herein are directly applicable to Java applications as well.

The remainder of this paper is structured as follows. In Section 2, we give an overview of the Android platform, present an example application we will use for instrumentation in the remainder of the paper, and discuss some Android-specific aspects like application signatures. Section 3 is dedicated to high-level instrumentation using AspectJ while Section 4 focuses on Tracematches. In Section 5, we introduce Soot and its Jimple intermediate representation which is then used for manual instrumentation in Section 6. Section 7 concludes the paper.

2 Android Platform Overview

The Android platform is built as a stack with various layers running on top of each other [10]. Lower-level layers provide services to upper-level layers. The lowermost layer is built on a customized Linux system and its libraries. The Android middleware builds an abtraction between the operating system and the user-level application running on the very top of the architecture stack. In this tutorial, we concentrate on instrumenting user-level applications.

Applications are provided to the user via different application markets like the official Google Play Store [11] and various third-party stores. Application developers can also host applications for download on their own websites.

2.1 Application Architecture

Most of the applications are written in the Java programming language[1]. They are compiled to Android's own bytecode format, called the Dalvik executable (dex). On application launch, the Android middleware spawns a new Dalvik Virtual Machine to execute the application's dex file. This enables Android to exploit the process isolation mechanisms of the underlying Linux operating system and ensures that all applications are run inside their own isolated containers. Note that Android applications do not use Java's concept of security managers [12]. Instead, Android implements its own permission systems for certain sensitive function calls. Furthermore, the Dalvik Virtual Machine is not

[1] In this tutorial, we disregard portions written in native code or script languages.

stack-based like the Java VM, but register-based and optimized for resource-constrained mobile devices [13].

Android applications do not have a single entry-point, such as the *main* method in Java. Developers must instead design the application in terms of components, each one adhering to a set of predefined interfaces. Every component is implemented as Java class derived from a specific base class in the Android middleware. Components react to OS events by overwriting the respective methods or calling specific OS methods to register further callbacks that are invoked when e.g. device's physical location changes.

There exist four different kinds of components: *activities*, *services*, *content providers*, and *broadcast receivers* [14]. *Activities* are single focused activities a user can interact with. They are the visible parts of an applications. In contrast, the *services* run in the background and are not interacting with the user directly. They are used for long-running background operations, such as MP3 playback. *Broadcast receivers* react to global events, such as incoming calls or text messages. *Content providers* implement domain-specific databases for, e.g., contacts [15].

The first three component types can communicate via asynchronous messages called *intents*. An intent is an abstract description of an action "intended" to be performed, such as *"launch the following website"*. Intents are a powerful feature in the Android platform that allow communication between components, both inside an application and across application boundaries. Intents are dispatched by the Android middleware, either to a directly specified receipient or to all receivers registered with the system for a specific intent type, e.g., all components capable of displaying a website to a user.

Each of the four different types of components have a distinct lifecyle that defines how the component is created, used and destroyed. The lifecycle is guided using events, i.e., a sequence of methods called by the OS. For instance, the `onCreate()` method gets called when an activity is loaded for the first time [16].

2.2 Android SMS Messenger Example

We next describe a simple Android application implementing an SMS Messenger. The app's user interface simply consists of two user inputs, one for the phone number and one for the message to be sent. When the user clicks on the"Send SMS" button, the application sends the given text message to the given phone number.

Listing 1.1 shows the corresponding source code. The code comprises the two methods `onCreate` and `sendSms`. As described in Section 2.1, the `onCreate` event method gets called when the activity is launched for the first time. The method defines some layout settings (`setContentView`) and prints out some debug information. Section 2.5 will give more details on Android's logging infrastructure.

The `sendSms` callback method is the more interesting part. It is called when the user clicks on the "Send SMS" button. The link between the method and the button is established using a layout XML file, which is a declarative way

```
1  public class RV2013 extends Activity
2  {
3    private EditText phoneNr, message;
4    private SmsManager smsManager = SmsManager.getDefault();
5
6    @Override
7    protected void onCreate(Bundle savedInstanceState) {
8      super.onCreate(savedInstanceState);
9      setContentView(R.layout.activity_rv2013);
10
11     Log.i("INFO", "in onCreate");
12 }
13
14 public void sendSms(View v){
15   Log.i("INFO", "in sendSms");
16
17   phoneNr = (EditText)findViewById(R.id.phoneNr);
18   message = (EditText)findViewById(R.id.message);
19
20   System.out.println("in sendSms");
21
22   smsManager.sendTextMessage(phoneNr.getText().toString(), null,
       message.getText().toString(), null, null);
23   }
24 }
```

Listing 1.1. Source Code of SMS Messenger Example

to register callsbacks for UI components[2]. The button handler again writes out some debug information ("in send Sms"), then extracts the user input in the different text fields using the *findViewById* OS function, afterwards calls the `println` method in the `PrintStream` class with the string "in SendSms" and finally sends out the SMS message, again using an OS function.

2.3 Overview of Android API Calls

The Android middleware provides abstractions for conveniently using device functions like sending SMS messages directly from applications written in Java without having to directly interact with native code libraries on the system level. The most important Android API methods for this paper are the following ones:

- `Log.i(String tag, String msg)`:
  Static methods which writes an `info` message to the log. The log can be browsed using the tool LogCat (c.f. Section 2.5).

  `tag`: usually identifies the class or activity where the log call occurs
  `msg`: the message that should be logged

- `findViewById(int id)`:
  Returns references to GUI objects. In this example, the `findViewById` method is used to get the text field contents for message text and recipient phone number. Note that such calls are generated by the compiler; one usually use the constants in the `R` pseudo class to access GUI elements from app code.

[2] The other alternative would have been to programatically set a listener in *onCreate*.

`id`: the object id to search for

- `SMSManager.sendTextMessage(String destinationAddress, String scAddress, String text, PendingIntent sentIntent, PendingIntent deliveryIntent)`:
  sends a text-based SMS message

  `destinationAddress`: the address to send the message to
  `scAddress`: the service center address; pass `null` to use the default
  `text`: the body of the message to send
  `sentIntent`: a broadcast message to be generated by the system when the message has been sent; pass `null` if not required
  `deliveryIntent`: a broadcast message for the message delivery; pass `null` if not required

2.4 Android.jar: Where Android Lives

The Android middleware consists of predefined Java classes and a set of native libraries. To be able to compile Android apps that make use of this API on a desktop PC, the Java classes of the Android API must be present. Therefore, the Android SDK provides these classes in a file called *android.jar* in its *platforms* directory. There is one such file for every version of the OS. We recommend using the appropriate version of the JAR file since new APIs are added from time to time and old, deprecated ones are removed. The minimum compatible OS version specified in the application's manifest file is usually a good pick.

However, note that these JAR files can only be used to create a somewhat complete callgraph and points-to set in the user code, but they cannot be used to actually run the application. This is because for many methods they only contain stubs and no actual implementations. Stubbed methods just throw a *NotImplementedException*. Obtaining a full *android.jar* file from a real phone is possible, but not trivial, as the Android API is stored in a precompiled and optimized file format. In most cases, such complete JAR files are not needed anyway.

2.5 Useful Tools

The Android SDK provides a number of tools that support a developer during the developement of an Android application. For instance, debugging or running on an emulator is essential during the developement phase. Therefore, we will briefly introduce the two most important tools: Logcat and the Android emulator.

Logcat. Android's logging system provides a mechanism for collecting different kinds of log messages from various applications and system components. These logs can be easily viewed and filtered using the `logcat` tool which is built on top of Android's debug bridge `adb`. Logcat can directly be launched from the command line with `adb logcat`. It supports various settings for filtering and

formatting the output as explained in [17]. The Android eclipse plugin provides a more convenient graphical user interface to logcat.

A log entry can be produced by invoking the static methods in the `android.util.Log` class. For instance, the statement `Log.e("TAG", "Ooops")` creates an error line in the log. The `Log.e` method takes two parameters: The tag (first parameter) can be used for filtering and categorization. The error message (second parameter) contains the error message or failure reason.

Android Emulator. As the name already suggests, the Android emulator [18] is a virtual mobile device that runs on a computer and is similar to a real device. It does not contain all the features of a real mobile device such as sending emails, but for most purposes, it is sufficiently complete. However, note that applications may suffer from serious performance penalties when run on the emulator.

With the help of the Android debug bridge [19], it is easy to create a new emulator. The command `android create avd -n <name> -t <targetID>` creates a new virtual device with the given name. The targetID is the API level one needs, e.g., 17 for Android 4.2. The new emulator is started by `emulator -avd <name>`. Afterwards, the virtual device's user interface is shown and one can interact with the emulator through the SDK's command-line tools. A more convenient way for the creation of an emulator is the usage of the grafical interface provided by eclipse (*Android Virtual Device Manager*).

2.6 Managing APKs on the Device

For author identification purposes, the Android framework requries that each application has to be signed with a certificate. This, for instance, allows the system to check whether an application update actually comes from the original application developer. Furthermore, applications signed with the same key are granted special privileges in interprocess communication. On the Android platform, it is common that most of the application certificates are self-signed [20]. When changing the APK file, e.g., by instrumenting the code, the signature is lost and the application must be signed again.

Standard tools like *keytool* and *jarsigner* can be used for signing the application. An example for the generation of a private/public key pair with *keytool* [20] is shown in Listing 1.2.

```
1 $ keytool -genkey -v -keystore my-release-key.keystore
2 -alias alias_name -keyalg RSA -keysize 2048 -validity 10000
```

Listing 1.2. Generation of a private/public key pair with keytool [20]

The jarsigner tool can be used for signing the application *my_application.apk* with the private key generated with keytool. Listing 1.3 shows the command for signing an app with jarsigner.

```
1 $ jarsigner -verbose -sigalg SHA1withRSA -digestalg SHA1 -keystore
2 my-release-key.keystore my_application.apk alias_name
```

Listing 1.3. Signing my_application.apk with jarsinger [20]

After signing the application, Google [21] recommends to use the *zipalign* tool to optimize the final APK. It ensures that all uncompressed data starts with a particular alignment relative to the start of the file. The zipalign tool comes with the Android SDK and Listing 1.4 shows the corresponding command.

```
$ zipalign -v 4 my_application.apk my_application_release.apk
```

Listing 1.4. Alignment of my_application.apk befor release

3 Instrumentation with abc and AspectJ

In this section, we describe how AspectJ and the AspectBench Compiler abc [9] can be used to declaratively instrument Android applications. Our goal is to modify the example application from Listing 1.1 such that no premium SMS messages to costly 0900 phone numbers can be sent anymore. Instead, an error message shall be written into the log file whenever the target phone number starts with 0900. SMS messages to normal phone numbers should be sent as usual. Obviously, this requires us to inline a monitor since no static analysis can know the target phone number the user is going to enter.

We create a new file *SendPremiumSMS.aj* with the contents shown in Listing 1.5. Note that the name of the file must match the name of the aspect. We first declare a pointcut for the `SmsManager.sendTextMessage` method. We could also have inlined it into the advice definition, but we are using it twice (once for blocking premium-rate SMS messages and once for logging that a message has actually been sent), so we keep it separate. The pointcut matches calls to the SMS sending method in the Android operating system, not our own user code. This way, we ensure that actually all SMS messages are intercepted which is especially useful when instrumenting unknown target applications.

```
import android.telephony.SmsManager;
import android.app.PendingIntent;
import android.util.Log;

public aspect SendSMS_PremiumAspect {
  pointcut sendSms(String no) :
     call(void SmsManager.sendTextMessage(..)) && args(no, ..);

  after(): sendSms(*) {
    Log.i("Aspect", "SMS message sent");
  }

  void around(String no): sendSms(no) {
    if (no.startsWith("0900"))
      Log.i("Aspect", "Premium SMS message blocked");
    else proceed(no);
  }
}
```

Listing 1.5. Aspect for blocking premium-rate SMS messages

In general, the aspects as such are written in the well-known AspectJ syntax and are not Android-specific except for the methods intercepted in the pointcuts and the ones called in the pieces of advice. All mapping to the Android platform is done by the abc compiler during the weaving process.

Since we only want a notification when an SMS message has actually been sent, the order of the pieces of advice inside the aspect is important: We place the *around* advice last to give it precedence over the *after* advice which shall only be executed when the *around* advice proceeds, i.e., the target phone number is not a premium-rate number. Otherwise, the SMS message is blocked and thus shall not be logged.

abc supports two different frontends for parsing Java source code: Polyglot and JastAdd. The Polyglot frontend is a bit dated and should not be used for instrumenting Android applications from source. JastAdd can be enabled as an extension using the `-ext abc.ja` command-line option. Also note that abc has its own class path which is independent of the JDK's class path and which must be set using the `-cp` option. It should include both the JRE's `rt.jar` file and abc's own `abc-runtime.jar` file. Since our target application references classes from the Android framework, we also need to include the `android.jar` file.

If applications written for modern versions of the Android API are also supported on older platforms (i.e., have a lower minimum SDK version that one they were developed for), the Android eclipse plugin automatically integrates so-called support classes which add some newer APIs to older platforms. The respective *jar* file can then found in the *libs* directory of the application project and needs to be included in abc's class path as well.

The complete command-line for instrumenting the example is shown in Listing 1.6. The Android support is enabled with the `-android` option, the APK file name is given with the `-injars` switch.

```
1 java -cp abc-ja-exts-complete.jar abc.main.Main \
2   -cp /path/to/rt.jar: \
3     /path/to/android-support-v4.jar: \
4     /path/to/android.jar: \
5     /path/to/abc-runtime.jar \
6   -ext abc.ja \
7   -android -injars /path/to/RV2013.apk \
8   /path/to/SendSMS_PremiumAspect.aj
```

Listing 1.6. abc compiler command-line

Unsigned applications will not run on the Android OS, which is why the instrumented *apk* file still needs to be signed before it can be run on a real phone or the emulator (c.f. Section 2.6).

4 Instrumentation with Tracematches

While AspectJ is rather convenient for describing Android instrumentations (see Section 3), it requires additional manual effort when sequences of actions shall be tracked. Tracematches [22] provide a simple regular-expression based approach to declaratively abstract from such tracking. Let us assume we want to raise an

```
import android.telephony.SmsManager;
import android.app.PendingIntent;

public aspect SMSSpam {
  tracematch(String no) {
    sym sendSms after:
      call (void SmsManager.sendTextMessage(..)) && args(no, ..);

    sendSms[3] sendSms+ {
      System.out.println("SMS spam detected to no: " + no);
    }

  }
}
```

Listing 1.7. Aspect for blocking premium-rate SMS messages

alert when more than three SMS messages are sent to the same phone number by an application, as this might indicate SMS spam. In AspectJ such counting would have to be implemented manually. In Tracematches, we simply define the pattern shown in Listing 1.7. Note that the name of the file and the name of the aspect must match, i.e., `SMSSpam.aj` in this case.

For compiling the tracematch, we again use the AspectBench compiler abc. The command-line is similar to the one shown in Section 3 for AspectJ, we only need to enable the tracematch extension as shown in Listing 1.8.

```
java -cp abc-ja-exts-complete.jar abc.main.Main \
  -cp /path/to/rt.jar: \
    /path/to/android-support-v4.jar: \
    /path/to/android.jar: \
    /path/to/abc-runtime.jar \
  -ext abc.ja.tm \
  -android -injars /path/to/RV2013.apk \
  /path/to/SMSSpam.aj
```

Listing 1.8. abc compiler command-line

5 The Machinery: Soot and Jimple

Soot [8] is an extensible program analysis and optimization framework for Java and Java-like environments such as Dalvik. It supports various input formats including Java source code, Java `class` files, and Dalvik `dex` files and also allows to write out these file formats after transformation. Figure 1 gives an overview of all possible input and output formats.

Code included in an Android application's *apk* file is automatically extracted before analysis. Afterwards, a new *apk* file containing the transformed code is built which can then be signed and executed on a phone or the emulator. abc uses Soot internally to weave aspects or tracematches into Java programs or Android apps.

Soot is organized in phases and packs [23]. Every pack contains an ordered list of phases. The first pack applied to every single method is the Jimple Bodies pack *jb* which translates the respective method's body into an intermediate representation called *Jimple*. Afterwards, if whole-program analysis is enabled, a number

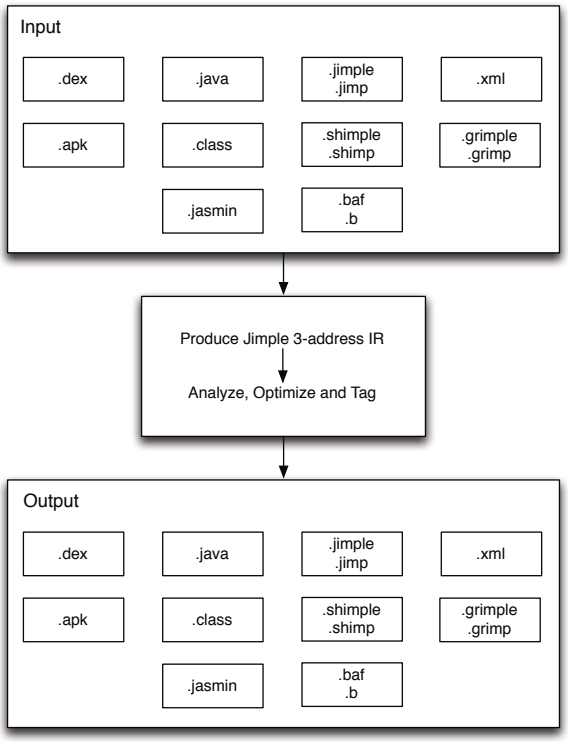

Fig. 1. Input and Output Formats in Soot

of whole-program packs run. They do not target single methods or classes, but the whole so-called *scene* containing all classes that have been loaded. Which classes are loaded depends on Soot's command line options. Consult the online documentation for details [24]. Usually, you only need to enable whole program mode if your analysis requires a complete call graph. If not, you can skip these phases by leaving the whole-program-mode option disabled which can considerably improve performance.

The first whole-program pack to run is the *cg* pack which creates the callgraph. Soot implements various callgraph construction algorithms. In this tutoral, we will use SPARK [25] for maximum precision. In some cases, less precise, but faster algorithms might be more appropriate. Once the callgraph is done, three more whole-program packs (whole-jimple-transformation, whole-jimple-optimization, whole-jimple-annotation) are executed, followed by a sequence of single method-packs (jimple-transformation, jimple-optimization, jimple-annotation).

For our purposes, we leave the whole-porgram-mode disabled and add a new phase to the jimple-transformation pack *jtp* which places our code directly after

the Jimple bodies are produced and before all other optimizations like dead code elimination run. This allows us to exploit the transformations done in the latter. If we needed a complete callgraph, we would use the whole-jimple-transformation-pack *wjtp* instead.

In the remainder of this section, we will show how to programatically configure and launch Soot, how to access the Jimple code of a method, and explain how Jimple is structured.

5.1 Jimple: Java, But Simple

Jimple stands between full Java sourcecode on one side and Java/Dalvik bytecode on the other side. While the first is impractically complex for static analysis or program transformations, the latter is quite cumbersome to work with because of its large number of (untyped) instructions. Jimple combines the advantages of both sides: There is only a limited instruction set, data is stored in variables, and statements are generally of a simple three-operand form. More complex statements or expressions are broken up into simple single-operation pieces and a set of intermediate variables. For instance, in Jimple `a=b+c+2` would be transformed to `temp=b+c, a=temp+2` with a new intermediate variable `temp`.

Jimple contains two general concepts: *locals* which are local variables and *units* which are statements. Every method body contains one chain of locals and one ordered chain of units. Units are usually of some type derived from *Stmt*, which in turn can contain references to expressions derived from *Expr*. Jimple generalizes all Java constructs to units and locals. The Java *this* reference, for instance, is assigned to a local at the beginning of an instance method. Afterwards, it behaves just like an ordinary local variable. The same happens with method parameters. These special assignments are called *IdentityStatement*s (c.f. lines 12 and 13 in Listing 1.9).

Assignments between locals, constants, and fields are done using *AssignStatement*s (c.f. line 23). Since Soot represents the AST as an object model in memory, the left and right side of an assignment are references to the objects representing the expressions standing on either side. Programatically traversing Jimple code thus simply means following chains of references. For instance, in line 16, the right-hand side of the assignment is a typecast represented by a *CastExpr* object.

To call methods, Jimple supports four different expressions, depending on the type of the target method. The three most important ones are *VirtualInvokeExpr* for a virtual dispatch invoke to an instance method (lines 15 and 18), *StaticInvokeExpr* for calling a static method (line 14), and *InterfaceInvokeExpr* for calling a method of an object of which only its interface type is known (line 26). Any invoke expression can be part of standalone statement called *InvokeStmt* (lines 14, 22, and 30), but can also serve as the right side of an assignment (i.e. *AssignStmt*, e.g., in line 15) unless the return type is `void`.

```
1  public void sendSms(android.view.View)
2  {
3    de.ecspride.RV2013 $r0;
4    android.view.View $r1;
5    java.lang.String $r2, $r3;
6    android.widget.EditText $r4;
7    int $i0;
8    java.io.PrintStream $r5;
9    android.telephony.SmsManager $r6;
10   android.text.Editable $r7;
11
12   $r0 := @this: de.ecspride.RV2013;
13   $r1 := @parameter0: android.view.View;
14   staticinvoke <android.util.Log: int
       i(java.lang.String,java.lang.String)>("INFO", "in sendSms");
15   $r1 =  virtualinvoke $r0.<de.ecspride.RV2013: android.view.View
       findViewById(int)>(2131165184);
16   $r4 = (android.widget.EditText) $r1;
17   $r0.<de.ecspride.RV2013: android.widget.EditText phoneNr> = $r4;
18   $r1 = virtualinvoke $r0.<de.ecspride.RV2013: android.view.View
       findViewById(int)>(2131165187);
19   $r4 = (android.widget.EditText) $r1;
20   $r0.<de.ecspride.RV2013: android.widget.EditText message> = $r4;
21   $r5 = <java.lang.System: java.io.PrintStream out>;
22   virtualinvoke $r5.<java.io.PrintStream: void
       println(java.lang.String)>("in sendSms");
23   $r6 = $r0.<de.ecspride.RV2013: android.telephony.SmsManager
       smsManager>;
24   $r4 = $r0.<de.ecspride.RV2013: android.widget.EditText phoneNr>;
25   $r7 = virtualinvoke $r4.<android.widget.EditText: android.text.Editable
       getText()>();
26   $r3 = interfaceinvoke $r7.<android.text.Editable: java.lang.String
       toString()>();
27   $r4 = $r0.<de.ecspride.RV2013: android.widget.EditText message>;
28   $r7 = virtualinvoke $r4.<android.widget.EditText: android.text.Editable
       getText()>();
29   $r2 = interfaceinvoke $r7.<android.text.Editable: java.lang.String
       toString()>();
30   virtualinvoke $r6.<android.telephony.SmsManager: void
       sendTextMessage(java.lang.String,java.lang.String,java.lang.String,
             android.app.PendingIntent,android.app.PendingIntent)>($r3,
       null, $r2, null, null);
31   return;
32 }
```

Listing 1.9. Jimple code for sending an SMS message

Method bodies are commonly analyzed by iterating over the units (i.e., statements) they comprise. For program rewriting, the chain of units is patched by removing existing units, inserting new units at the desired positions, or changing the expressions within existing units. All of these changes will be explained in the remainder of this paper.

5.2 Soot Options

Soot provides a lot of different command-line options. An online tutorial [26] gives a good overview of the different kinds of options available. The most important options for instrumenting Android applications are the following:

`-cp` *`pathlist`*: The classpath to be used when loading classes into Soot. Not to be confused with the classpath used by the JVM.

`-pp`: This option prepends the VM's classpath to Soot's own classpath.
`-validate`: Causes sanity checks to be performed on Jimple bodies to make sure the transformations have caused no type errors. This option may degrade Soot's performance, but might be useful for debugging instrumentation code.
`-output-format` *`format`*: Specifies the format of output files Soot should produce, if any. In case of Android instrumentation, the *dex* format has to be set. For debugging purposes, one can use the *jimple* output format to inspect the instrumentation results in the intermediate language. Note, though, that one cannot create outputs in multiple formats at the same time.
`-process-dir` *`dirs`*: Adds all classes in `dirs` to the set of classes to be analyzed and transformed by Soot. The list `dirs` can also contain *jar* or *apk* files.
`-src-prec` *`format`*: Sets format as Soot's preference for the type of source files to read when it looks for a class. In the case of Android, the *apk* format must be set.
`-w`: Tells Soot to enable the whole-program transformation packs. Required if one requires a callgraph or wants to use the *wjtp* pack for performing global transformations spanning multiple methods.
`-allow-phantom-refs`: Allows Soot to model classes not found on the classpath by stubs containing no methods or fields. Useful for saving memory by not including full implementations of some libraries.

These options can either be set via the command line or directly in the Java code via `Options.v()`, e.g., `Options.v().set_whole_program(true)` for enabling whole-program mode if required. Listing 1.10 shows an example of a possible Soot initialization for instrumenting Android applications.

```
1  private static boolean SOOT_INITIALIZED = false;
2  private final static String androidJAR = "./lib/android.jar";
3  private final static String apk = "./apk/RV2013.apk";
4
5  public static void initialiseSoot(){
6      if (SOOT_INITIALIZED)
7        return;
8
9      Options.v().set_allow_phantom_refs(true);
10     Options.v().set_prepend_classpath(true);
11     Options.v().set_validate(true);
12
13     Options.v().set_output_format(Options.output_format_dex);
14     Options.v().set_process_dir(Collections.singletonList(apk));
15     Options.v().set_force_android_jar(androidJAR);
16     Options.v().set_src_prec(Options.src_prec_apk);
17
18     Options.v().set_soot_classpath(androidJAR);
19
20     Scene.v().loadNecessaryClasses();
21
22     SOOT_INITIALIZED = true;
23 }
```

Listing 1.10. Soot Initialization Example for Instrumenting Android Applications

6 Manual Instrumentation

Besides the convenient way of instrumenting Android applications with the help of AspectJ (c.f. Section 3) or tracematches (c.f. Section 4), one can also use Soot to directly manipulate an Android application's code using the Jimple intermediate representation. This is especially important for instrumentations that are not possible with ApsectJ or tracematches. We described a range of such policies in previous work [27]. As a simple example, AspectJ cannot be used to remove debugging outputs including all intermediate computations that are only used in such debugging statements. These computations will remain even if the debug outputs as such are filtered using an *around* advice.

This section demonstrates the two possiblities in direct code modification: removing or adding code. Manipulating existing units is straight-forward given that knowledge. Instead of generating new Jimple units, one simply changes the fields of existing objects. As a first step, we need to configure launch Soot as desribed in section 5. We then register a *jimple-transformation* transformation phase as shown in listing 1.11. As discussed in section 5.2, this is more efficient than using a *whole-jimple-transformation* whole-program phase. Furthermore, we do not need a complete callgraph for our goal, so there is no reason to have Soot create one.

```
1 PackManager.v().getPack("jtp").add(
2     new Transform("jtp.myAnalysis", new MyBodyTransformer()));
3 PackManager.v().runPacks();
4 PackManager.v().writeOutput();
```

Listing 1.11. Adding new Phase to Jimple Transformation Pack

The `runPacks()` methods triggers the execution of the packs and calls the overwritten `internalTransform()` method inside the `MyBodyStranformer` class derived from `BodyTranformer`[3]. In order to iterate over all classes and methods in the Android application, one can use the code in listing 1.12 as a starting point. The code must be placed inside `internalTransform()`.

```
1  for (SootClass c : Scene.v().getApplicationClasses()) {
2    for(SootMethod m : c.getMethods()){
3      if(m.isConcrete()){
4         Body body = m.retrieveActiveBody();
5         Iterator<Unit> i = body.getUnits().snapshotIterator();
6         while (i.hasNext()) {
7           Unit u = i.next();
8           //do something
9         }
10     }
11   }
12 }
```

Listing 1.12. Iterating over the Android Code

[3] In whole-program-mode, we would have used a *SceneTransformer* in the *wjtp* pack.

One important point in this code snippet is the `snapshotIterator()` method that should be used if statements in the method's body will be changed while the loop runs. This avoids `ConcurrentModificationException`s.

6.1 Removing Statements

The fact that all statements in the body of a method are stored into a chain makes it very easy to remove a complete statement from the Jimple code. This can be done by just removing it from the chain:

```
body.getUnits().remove(unit);
```

After removing statements, dead code can remain. Soot already offers optimizations for removing such code, propagating definitions that are only used once, and others. If the subsequent Jimple optimization pack (`jop`) is enabled, those optimizations are applied automatically.

6.2 Adding New Statements

In general, the unit chain allows new statements to be placed before (`insertBefore()`) or after (`insertAfter()`) a specific code point. These must be fully-constructed Jimple units containing all required expressions, i.e., the operands in case of a primitive arithmetic operation. The `Jimple.v()` singleton provides factory methods called `Jimple.v().newX` for generating new Jimple statments and expressions where `X` stands for the different kinds of AST elements. An example is `newStaticInvokeExpr()` which creates a new static invoke expression to be used inside an invoke statement or as the right side of an assignment.

Let us go back to our original SMS Messenger Example (c.f. Section 2.2) and insert some checks that prevent the application from sending SMS messages to premium rate numbers. This check has to be placed before the `sendTextMessage()` method in line 30 and could look like the one described in Listing 1.13 for premium rate numbers that start with 0900.

```
1 if(!phoneNr.getText().toString().startsWith("0900"))
2    smsManager.sendTextMessage(phoneNr.getText().toString(), null,
3      message.getText().toString(), null, null);
```

Listing 1.13. 0900 Premium Rate SMS Check

Before this statement can be constructed, various expressions must be generated:

- String constant for the number "0900"
- Method call to the `startsWith()` method. The result must be stored in a new local variable that does not conflict with any existing local variable.
- "if" statement with *then* and *else* branch

A complete example for integrating such a check is described in Listing 1.14.

```
1  private void eliminatePremiumRateSMS(Unit u, Body body) {
2    Stmt stmt = (Stmt) u;
3    if (stmt.containsInvokeExpr()){
4      InvokeExpr iinv = (InvokeExpr) invoke.getInvokeExpr();
5      if(iinv.getMethod().getSignature().equals(SEND_SMS_SIGNATURE)){
6         Value phoneNumber = invoke.getInvokeExpr().getArg(0);
7         if (phoneNumber instanceof Local){
8           Local phoneNoLocal = (Local)phoneNumber;
9
10          // Invoke startsWith and save result
11          VirtualInvokeExpr inv = generateStartsWith(body, phoneNoLocal);
12          Local invRes = generateNewLocal(body, BooleanType.v());
13          AssignStmt astmt = Jimple.v().newAssignStmt(invRes, inv);
14          body.getUnits().insertBefore(astmt, u);
15
16          //generate condition
17          NopStmt nop = Jimple.v().newNopStmt();
18          IfStmt ifStmt = Jimple.v().newIfStmt(invRes, nop);
19
20          body.getUnits().insertBefore(ifStmt, u);
21          body.getUnits().insertAfter(nop, u);
22        }
23     }
24   }
25
26 private InvokeExpr generateStartsWith(Body body, Local phoneNoLocal) {
27   SootMethod sm = Scene.v().getMethod(STARTS_WITH_SIGANTURE);
28   return Jimple.v().newVirtualInvokeExpr(phoneNoLocal, sm.makeRef(),
       StringConstant.v("0900"));
29 }
30
31 private Local generateNewLocal(Body body, Type type){
32   LocalGenerator lg = new LocalGenerator(body);
33   return lg.generateLocal(type);
34 }
```

Listing 1.14. Generation of Jimple Statements for Premium Rate SMS Check

`SEND_SMS_SIGNATURE` is a string constant containing the method signature of the *sendTextMessage*. `STARTS_WITH_SIGNATURE` is the signature of the `startsWith()` method in the `String` class.

Note that we do not directly create new locals by giving a name and a type. Instead, we defer this task to the `LocalGenerator` class which automatically creates a unique local name.

Finally, the `eliminatePremiumRateSMS()` method has to be called inside the code snipped shown in Listing 1.12 so that the instrumentation is performed for all methods that possibly send SMS messages.

7 Conclusion

In this tutorial paper, we have shown how to instrument Android applications using AspectJ, Tracematches and manual imperative instrumentation based on Soot. All these techniques can also be applied to classical Java programs. For Android, there are a number of platform-specific issues to keep in mind such as the need for signing the APK file before running it on a phone or the emulator.

The techniques shown in this paper can not only be used for security purposes, but also for code optimization and analysis in general. Many optimizations like constant propagation or dead code elimination are already built into Soot, making instrumentations easier for the user.

8 Examples

The SMS Messenger example (RV2013) as well as the instrumentation examples can be downloaded from `https://github.com/secure-software-engineering/android-instrumentation-tutorial`

References

1. International Data Corporation: Worldwide quarterly mobile phone tracker 3q12 (November 2012), `http://www.idc.com/tracker/showproductinfo.jsp?prod_id=37`
2. Bit9: Pausing google play: More than 100,000 android apps may pose security risks (November 2012), `http://www.bit9.com/pausing-google-play/`
3. Lu, L., Li, Z., Wu, Z., Lee, W., Jiang, G.: Chex: statically vetting android apps for component hijacking vulnerabilities. In: Proceedings of the 2012 ACM Conference on Computer and Communications Security, pp. 229–240. ACM (2012)
4. Kim, J., Yoon, Y., Yi, K., Shin, J., Center, S.: Scandal: Static analyzer for detecting privacy leaks in android applications. In: Proceedings of the Workshop on Mobile Security Technologies (MoST), in Conjunction with the IEEE Symposium on Security and Privacy (2012)
5. Yang, Z., Yang, M.: Leakminer: Detect information leakage on android with static taint analysis. In: IEEE 2012 Third World Congress on Software Engineering (WCSE), pp. 101–104 (2012)
6. Enck, W., Gilbert, P., Chun, B.G., Cox, L.P., Jung, J., McDaniel, P., Sheth, A.N.: Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In: Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation. OSDI 2010, pp. 1–6. USENIX Association, Berkeley (2010)
7. Xu, R., Saïdi, H., Anderson, R.: Aurasium: practical policy enforcement for android applications. In: Proceedings of the 21st USENIX Conference on Security Symposium, Security 2012, pp. 27–27. USENIX Association, Berkeley (2012)
8. Lam, P., Bodden, E., Lhoták, O., Hendren, L.: The soot framework for java program analysis: a retrospective. In: Cetus Users and Compiler Infastructure Workshop, CETUS 2011 (October 2011)
9. Allan, C., et al.: Abc: the aspectbench compiler for aspectj. In: Glück, R., Lowry, M. (eds.) GPCE 2005. LNCS, vol. 3676, pp. 10–16. Springer, Heidelberg (2005)
10. Android: Android security overview (December 2012), `http://source.android.com/tech/security/`
11. Goolge Inc.: Google play (December 2012), `https://play.google.com/`
12. Bodden, E., Hermann, B., Lerch, J., Mezini, M.: Reducing human factors in software security architectures. In: Future Security Conference (to appear, September 2013)
13. Oh, H.S., Kim, B.J., Choi, H.K., Moon, S.M.: Evaluation of android dalvik virtual machine. In: Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems, JTRES 2012, pp. 115–124 (2012)

14. Google Inc.: Application fundamentals (December 2012), http://developer.android.com/guide/components/fundamentals.html
15. Google Inc.: Content provider basics (December 2012), http://developer.android.com/guide/topics/providers/content-provider-basics.html
16. Google Inc.: Activity (June 2013), http://developer.android.com/reference/android/app/Activity.html
17. Google Inc.: Logcat (June 2013), http://developer.android.com/tools/help/logcat.html
18. Google Inc.: Android emulator (June 2013), http://developer.android.com/tools/help/emulator.html
19. Google Inc.: Android debug bridge (June 2013), http://developer.android.com/tools/help/adb.html
20. Google Inc.: Signing your applications (June 2013), http://developer.android.com/tools/publishing/app-signing.html
21. Google Inc.: zipalign (June 2013), http://developer.android.com/tools/help/zipalign.html
22. Allan, C., Avgustinov, P., Christensen, A.S., Hendren, L., Kuzins, S., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G., Tibble, J.: Adding trace matching with free variables to aspectj. In: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications. OOPSLA 2005, pp. 345–364. ACM, New York (2005)
23. Bodden, E.: Packs and phases in soot (November 2008), http://www.bodden.de/2008/11/26/soot-packs/
24. Lam, P., Qian, F., Lhoták, O.: Packs and phases in soot (November 2008), http://www.sable.mcgill.ca/soot/tutorial/phase/
25. Lhoták, O., Hendren, L.: Scaling java points-to analysis using spark. In: Hedin, G. (ed.) CC 2003. LNCS, vol. 2622, pp. 153–169. Springer, Heidelberg (2003)
26. Patrick Lam, F.Q., Lhoták, O.: Soot command-line options (June 2013), http://www.sable.mcgill.ca/soot/tutorial/usage
27. Arzt, S., Falzon, K., Follner, A., Rasthofer, S., Bodden, E., Stolz, V.: How useful are existing monitoring languages for securing android apps? In: 6. Arbeitstagung Programmiersprachen (ATPS 2013). Lecture Notes in Informatics, Gesellschaft für Informatik (February 2013)