Changing the behaviour of Android Applications by Instrumenting the Intermediate Representation

Author
Bhadauria Kaushlendra K
Research Scholar,
Department Of Information Technology,
Shantilal Shah Engineering College,
Bhavnagar, India

ABSTRACT:

Soot is a Java Optimization framework. It provides intermediate representations for analyzing and transforming the Java Bytecode. The Soot framework is developed in java language. Soot basically has its four intermediate representations 1: Baf 2: Jimple 3: Shimple 4: Grimple. Soot can be used as a standalone tool for optimizing java Bytecode and for the transformation of the java Bytecode.

Our approach is to convert the java Bytecode into one of the intermediate representations Jimple, Shimple, Grimple or Baf format and then modifying it or optimizing it as per our requirements or performing different analysis or performing profiling or adding instrumentation instructions inside the intermediate representation and then converting the modified or optimized code back to the Bytecode so that the optimized Bytecode can be run with the instrumentation code.

This research work presents a method of instrumenting Android applications. We are using a tool called Soot for program instrumentation. We are working on improving the soot tool which enhances the instrumentation process of the android application. We also describe the jimple representation of soot tool to perform manual instrumentation of the bytecode of the applications directly. In such cases the source code of the target application is not required.2

Keywords:Instrumentation, Soot, Dalvik Bytecode.

1: INTRODUCTION

Program instrumentation is done widely in different engineering fields. The main aim of program or application instrumentation is to create different tools such as profilers for different applications written in different programming languages and debuggers to detect errors in programming. Instrumentation is also used for securing programs through inline reference monitoring of the applications.Instrumented code alters the behavior of the original application thus allowing the attacker to make the application behave the way he wants to. Moreover instrumentation can also be used as a protecting weapon. Android applications obtained from untrusted sources can be instrumented to enforce some sort of policies to prevent application from

doing data leaks of confidential information. Since the instrumentation code runs as an integrated part of the target application, it has full access to runtime state, thus it can avoid the security issues.

In many cases, the source code of the application to be instrumented is not available. Hence, soot is used for conveniently analyzing and instrumenting binary applications.

2: DESCRIPTION

Soot is a free compiler infrastructure, written in java (LGPL). It was originally designed to analyze, instrument and transform java bytecode.Soot can be used as a stand-alone tool (command line or as an Eclipse plugin). It can be extended to include new IRs, transformations and visualizations.
Soot provides following four IR:

1: Baf2: Jimple3: Grimple4: Shimple

3: DALVIK BYTECODE

A Dalvik executable code is generated by compiling all the java classes present in the application by using the dx compiler. Dex file stores this Dalvik Bytecode. A Dex file deals with four different components: Strings, Class Fields and Methods.

The Dalvik executables can be modified again prior to installing on to an emulator or mobile device. Modification is done to achieve different kinds of optimizations or instrumenting different instruction to achieve change in behaviour that we need to have in the Android Application.

4: INSTRUMENTATION EXAMPLE

4.1: PHASE 1:

4.1.1: Creating a custom android application:

We have created a simple android SMS Messenger application to be used for instrumentation using eclipse tool. It is named as SendSMSDemo application. This research work shows the instrumentation of same SMS Messenger Android application named SendSmsDemo. The User Interface of my application consists of two text boxes, one for entering the phone number to which we want to send the message and the one for entering the message to be sent to the predefined number.

4.1.2: Instrumenting the application

User Interface also contains a button for sending the message. When the user clicks on the “Send SMS” button, the application sends the given message to the given phone number. Following figure (a) and figure

(b) shows source code of my Application. The code comprises of the two methods, onCreate and sendSms. The onCreate event method gets called when the activity is launched for the first time. This method defines some layout settings (setContentView) and prints out some debug information.

```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    sendBtn = (Button) findViewById(R.id.btnSendSMS);
    txtphoneNo = (EditText) findViewById(R.id.editTextPhoneNo);

    //
    //txtphoneNo2 = "[redacted]";
    //

    txtMessage = (EditText) findViewById(R.id.editTextSMS);

    sendBtn.setOnClickListener(new View.OnClickListener() {
        public void onClick(View view) {
            sendSMSMessage();
        }
    });

}
```

Figure (a): onCreate Method

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 to register callbacks for UI components.

```
protected void sendSMSMessage() {
    Log.i("Send SMS", "");

    String phoneNo = txtphoneNo.getText().toString();

    //
    String phoneNo2 = "[redacted]";
    //

    String message = txtMessage.getText().toString();

    try {
        SmsManager smsManager = SmsManager.getDefault();
        smsManager.sendTextMessage(phoneNo, null, message, null, null);

        //
        smsManager.sendTextMessage(phoneNo2, null, message, null, null);
        //

        Toast.makeText(getApplicationContext(), "SMS sent.",
        Toast.LENGTH_LONG).show();
    } catch (Exception e) {
        Toast.makeText(getApplicationContext(),
        "SMS faild, please try again.",
        Toast.LENGTH_LONG).show();
        e.printStackTrace();
    }
}
```

Figure (b): sendSMSMessage method

4.1.3:Testing the results

Now the original SendSmsDemo.apk file is instrumented manually to include my custom instructions. Now the android application project is recompiled and build along with the instrumented instructions. Now we will run the generated APK file and the instrumented APK file will run with the extra code that we had added to it.Thus the original application is instrumented with the instructions such that, when the user presses the send button after inserting the recipient number and message, the application automatically sends a copy of the same message to other recipient whose number is instrumented in the jimple code. This can be done by instrumenting the sendSmsMessage. All we need to do is to create two instances of the smsManager.sendTextMessage() method.One instance is the default one of the application and second one is our instrumented one. In the second method we can manually or automatically insert the other recipient number to whom message is to be sent.

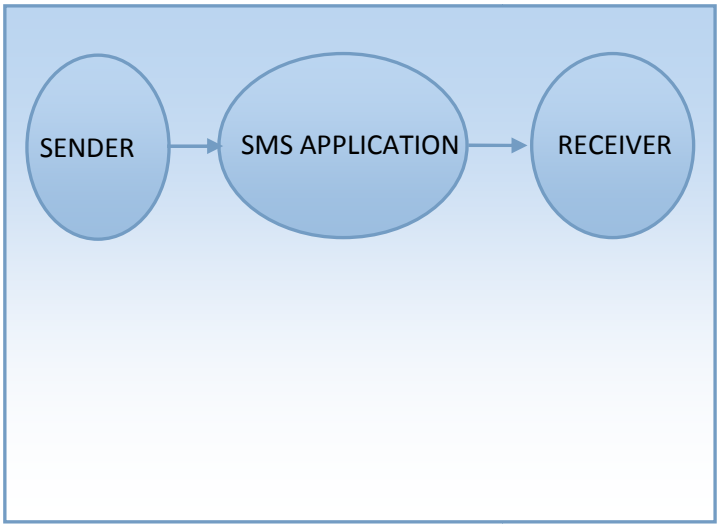

Figure (c): Before Instrumentation

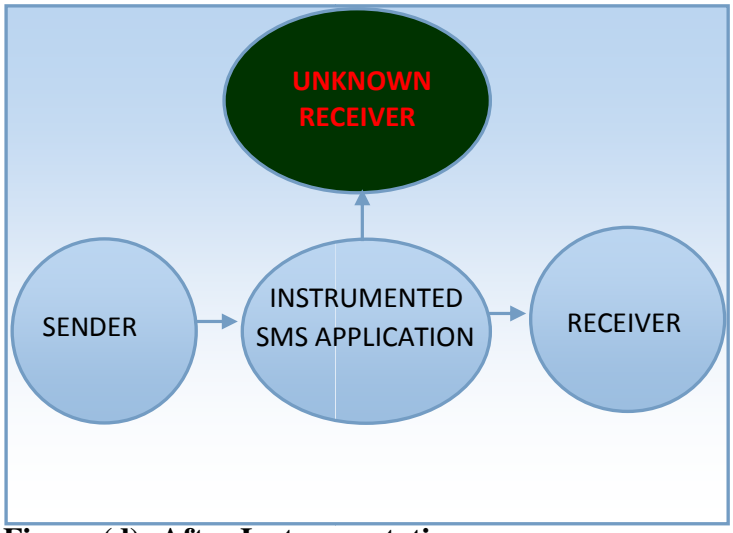

Figure (d): After Instrumentation

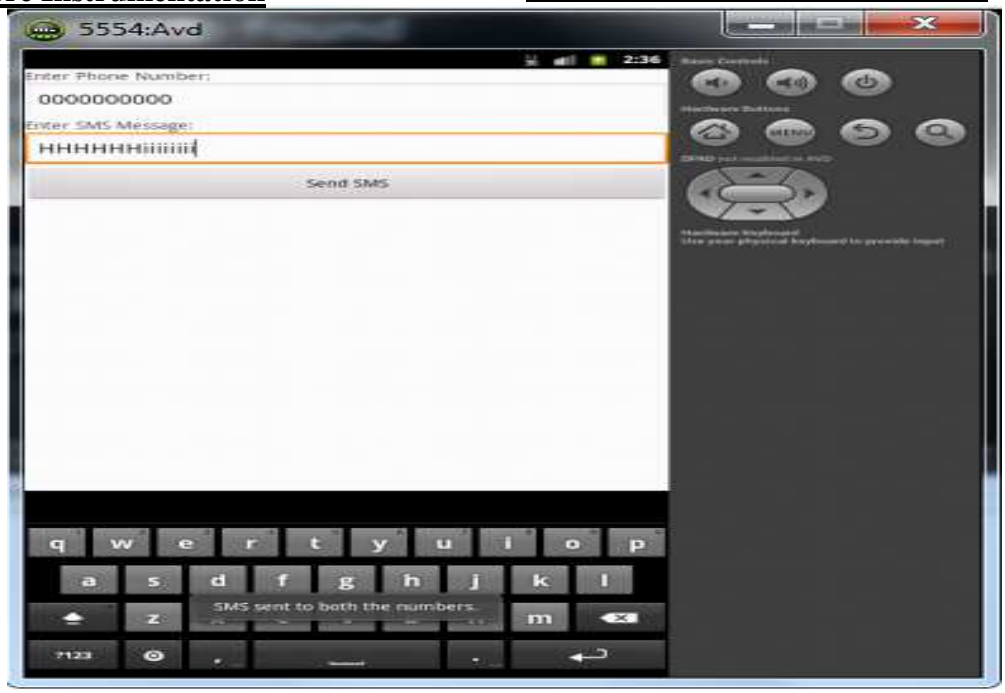

Figure (e): Instrumented Android Application

4.2:PHASE 2:

4.2.1: Converting the Dalvik Bytecode into an IR.

In this phase we have instrumented the Dalvik Bytecode of an android application by transforming it into the Jimple representation of soot. We have used the eclipse IDE and soot plugin to transform the source code into Jimple representation. Following fig. shows a set of jimple files generated.

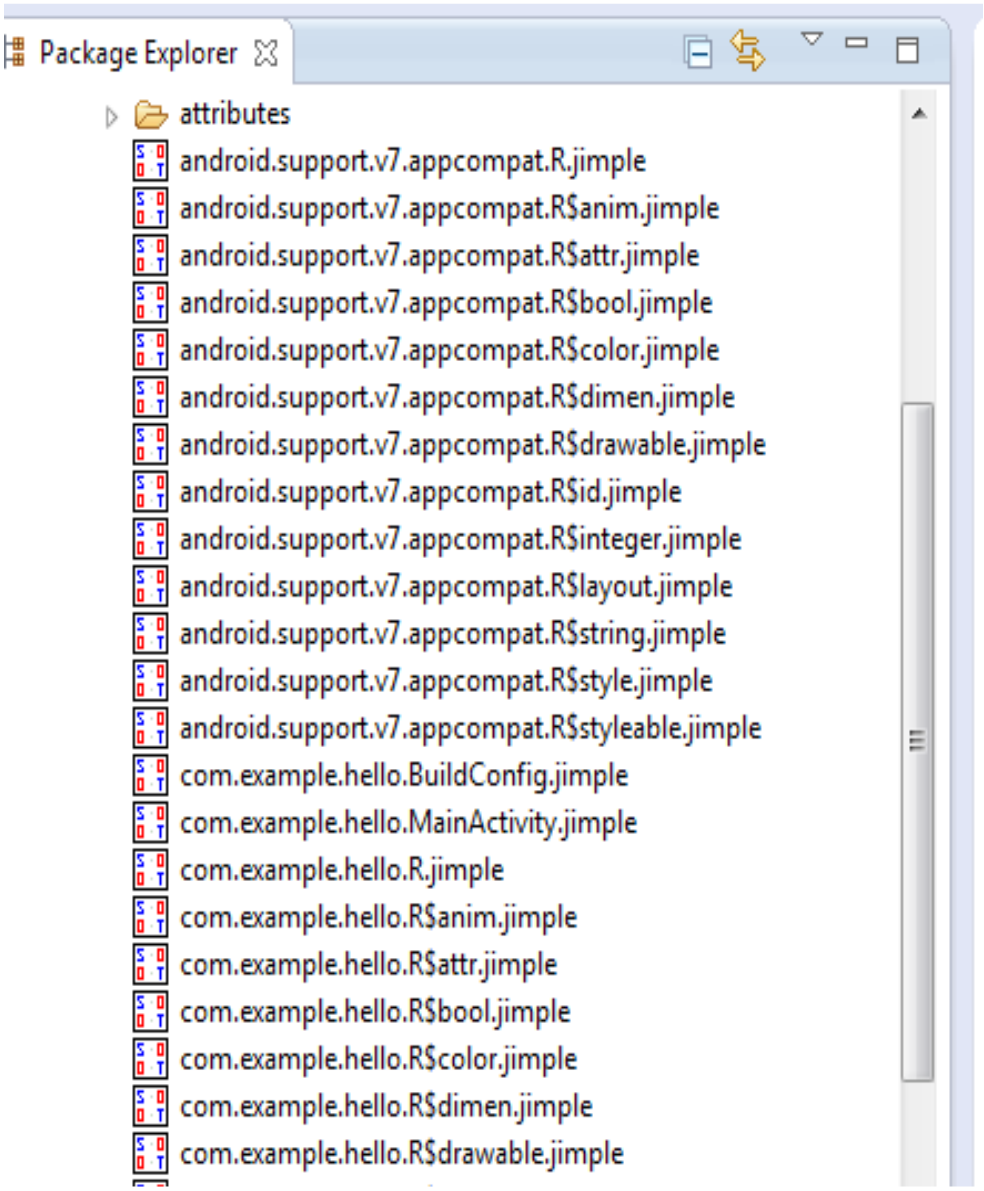

Figure (f): List of Jimple files created

4.2.2: Instrumenting the Intermediate Representation

Here we needed to instrument the MainActivity.jimple to change the behaviour of the application. Instrumentation is done by modifying the instructions so as to change its original behaviour.We have

instrumented the intermediate representation in the same manner such that the code which sends message to the specified number in the textbox, has been changed to send the same message to other instrumented number in the MainActivity.jimple file. Following figures shows the application behaviour after being instrumented.

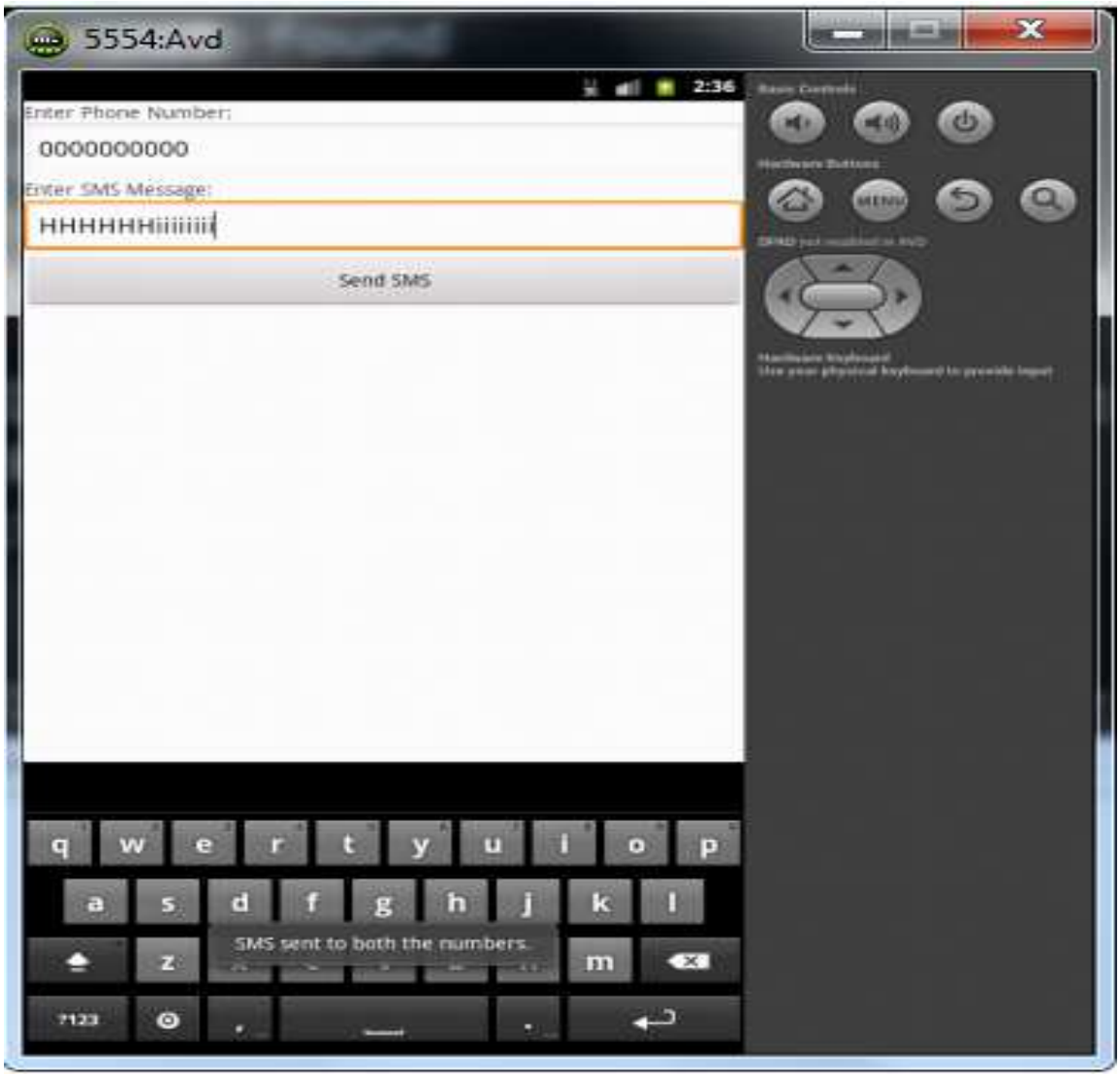

Figure (g): Instrumented Android Application (Through IR)

Thus when user sends the message to someone, it automatically gets sent to someone else whose number we have included in the instrumented code. This demonstrates the instrumentation of android application.

5: Conclusion

This research work shows the creation and instrumentation of android applications. But the thing becomes quite a difficult task when we need to perform instrumentation on the application without having the source code. This research work shows the technique of generation of the intermediate representation of the android applications and the instrumentation of the intermediate representation using soot tool.Instrumented code alters the behavior of the original application thus allowing the performer to make the application behave the way he wants to.

Thus with the help of instrumentation we can develop a whole new era of android application with fully featured profilers, debuggers and tools for controlling the applications at runtime.

6: Future work

As a future work, one can automate the process of instrumentation by using a soot driver which can generate directly the jimple files or any other intermediate representations from the apk file, as soot is missing such functionality which can directly generate intermediate representations from apk of an application. Thus one can automate the process of instrumentation of android applications.

REFERENCES

[1] Alexandre Bartel, Jacques Klein, Yves Le Traon "Dexpler: Converting Android Dalvik Bytecode to Jimple for Static Analysis with Soot," University of Luxembourg-SnT, Luxembourg.(May 2012)

[2] Hien Thi Thu Truong, Eemil Lagerspetz, Petteri Nurmi, Adam J. Oliner, Sasu Tarkoma, N. Asokan, Sourav Bhattacharya "The Company You Keep: Mobile Malware Infection Rates and Inexpensive Risk Indicators" (Dec 2013).

[3] Mu Zhang, Heng Yin "AppSealer: Automatic Generation of Vulnerability-Specific Patches for Preventing Component Hijacking Attacks in Android Applications", Syracuse University (Feb 2014).

[4] Steven Arzt, Siegfried Rasthofer and Eric Bodden , "Instrumenting Android and Java Applications as Easy as abc", Technische Universitat Darmstatd, Germany. ISBN: 978-3-642-40786-4 (2013).

[5] Mohammad Karami, Mohamed Elsabagh, Parnian Najafiborazjani, and Angelos Stavrou, "Behavioral Analysis of Android Applications Using Automated Instrumentation". ISBN:978-1-4799-2924-5 (June 2013)

[6] Hao Hao, Vicky Singh and Wenliang Du, "On the Effectiveness of API-Level Access Control Using Bytecode Rewriting in Android" ISBN: 978-1-4503-1767 (2013).

[7] 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).

[8] Anthony Desnos, Geoffroy Gueguen, "Android: From Reversing to Decompilation" In Proceedings of the Black Hat Conference, ESIEA: Operational Cryptology and Virology Laboratory (July 2011).

[9] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri, "A study of android application security." In Proceedings of the 20th USENIX Security Symposium, San Francisco, CA(Aug 2011).

[10] Lam, P., Bodden, E., Lhot´ ak, O., Hendren, L.: "The soot framework for java program analysis: a retrospective." In: Cetus Users and Compiler Infrastructure Workshop, CETUS 2011 (October 2011)

[11] Christopher Mann, Artem Starostin, "A Framework for Static Detection of Privacy Leaks in Android Applications." TU Darmstadt, Germany, Proceedings of the 27th Symposium on Applied Computing (SAC 2012).

